UCLouvain

École polytechnique de Louvain

Bringing PETs home: A Dataflow Programming Model and Runtime for Privacy-Preserving Smart Home Applications

Authors: **Sacha Defrère, Henri Pihet**
Supervisor: **Pr. Etienne Riviere**
Readers: **Dr. Annanda Rath, Julien Gourgue**
Academic year 2025–2026
Master [120] in Computer Science

Contents

List of Acronyms

ABAC Attribute-Based Access Control.

ACL Access Control List.

DNN Deep Neural Network.

HyBAC Hybrid Access Control.

IoT Internet of Things.

LoRa Long Range (radio).

MJPEG Motion JPEG.

MQTT Message Queuing Telemetry Transport.

NUC Next Unit of Computing.

ONNX Open Neural Network Exchange.

PET Privacy-Enhancing Technology.

RBAC Role-Based Access Control.

TABAC Trigger-Action-Based Access Control.

TPU Tensor Processing Unit.

YOLO You Only Look Once.

Diagram Colour Conventions

The figures in this thesis use a consistent colour scheme to identify the role of each component:

- Hub and system components
- Connected devices and sensors
- Third-party applications and PET modules
- External cloud services

Chapter 1

Introduction

Consider a simple scenario: your doorbell rings. Your smart camera captures the moment, and within milliseconds the footage is uploaded to a cloud server operated by the device manufacturer, possibly in another country. The cloud processes the video, runs face recognition, and pushes a notification to your phone. Two seconds. The doorbell worked exactly as advertised.

At no point, however, did you decide what data left your home, where it went, or how long it would be stored. The camera did what it was designed to do, and that design happened to involve transferring your visitors' biometric data to a third party without your explicit consent, as illustrated on Figure 1.1.

This is not an exceptional case. It describes the standard architecture of the vast majority of smart home devices sold today. As homes fill with more connected objects, cameras, microphones, motion sensors, thermostats, door locks, even pet toys, the amount of intimate data continuously flowing out of the home grows with them.

This thesis proposes a different approach. Rather than accepting this state of affairs, we introduce a framework that gives users fine-grained, programmable

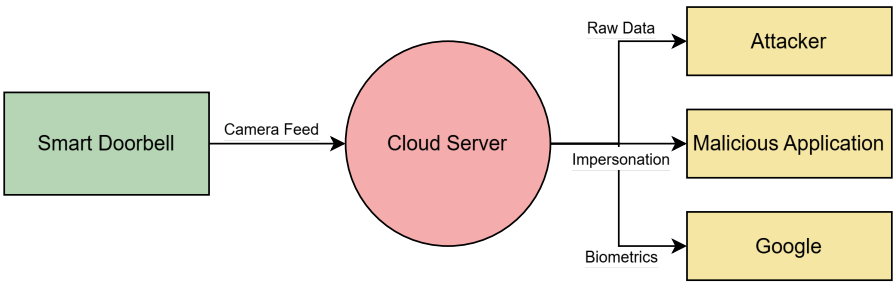


Figure 1.1: Typical example where data leaves the home without explicit user consent, exposing it to various risks

control over what data leaves their home, to whom it is sent, and in what form. The framework runs on a local hub, a dedicated device inside the home that mediates all data flows before they leave the local network. In the doorbell example, an application could be constrained to receive only a blurred version of the video stream, faces obscured on the hub before any data is transmitted. The application functions normally; the raw footage never leaves the house.

Other examples of what becomes possible: a living room monitoring service that receives only a high-level signal *person detected* or *no presence* rather than a continuous video feed. Or a voice assistant that processes speech locally and forwards only a scrambled audio signal to external servers, preventing the exposure of the user's raw voice. These are not hypothetical features; they are use cases we implement and demonstrate in this work.

What this work proposes and what it assumes. We build on HubOS [1], a local-first smart home operating system that processes data within the home by default. Our contribution is an extension of HubOS that introduces a richer model of privacy control: one based on data flows, context-aware rules, and modular privacy-preserving components called Privacy-Enhancing Technologies (PETs). Although PETs are technically capable of performing these transformations to ensure privacy, the problem is that no existing smart home platform integrates them into its rule or permission model: they are deployed as standalone tools, disconnected from the policies that govern data access. Closing that gap is the central contribution of this thesis.

Adopting this framework carries real constraints. Applications must declare their data flows explicitly. They must operate within a structured rule model that can restrict or modify what they receive. Their data may be transformed in transit by a PET before it reaches them. This is by design. The trade-off is deliberate: we accept these constraints in exchange for meaningfully stronger privacy guarantees. We also acknowledge that other smart home platforms exist, and we position our work relative to them throughout this document.

What this work does not address. We are building a privacy framework, not a hardened operating system. If an adversary already controls the hub at the OS level, our model does not protect against that. Physical security, OS hardening, and network-level intrusion prevention are important problems; they are not the problem we solve here. We do, however, describe the isolation mechanisms our system provides as a meaningful first layer of defence.

1.1 Challenges with Connected Devices

While cloud connectivity brings many useful features, it also introduces several important challenges that are often invisible to users.

Privacy concerns. Connected devices continuously collect data about users' behaviours, habits, and environments. Even when data is nominally anonymized, research has shown that combining multiple data sources can make it possible to re-identify individuals [2]. Beyond identification, data is routinely used for secondary purposes, analytics, behavioural profiling, or targeted advertising often without explicit user awareness or consent.

Without a privacy-aware system, users are left with a difficult choice: use the convenient cloud service and accept that their data will be handled in ways they cannot control, or opt out entirely and lose the functionality. We aim to break this false dichotomy. Our framework allows users to benefit from third-party applications while retaining meaningful control over the data those applications can access.

Security risks. Sending data to remote servers increases the attack surface. Vulnerabilities in cloud-connected devices have already been exploited in practice. In some cases, poorly secured systems allow unauthorized users to remotely access or control devices [3].

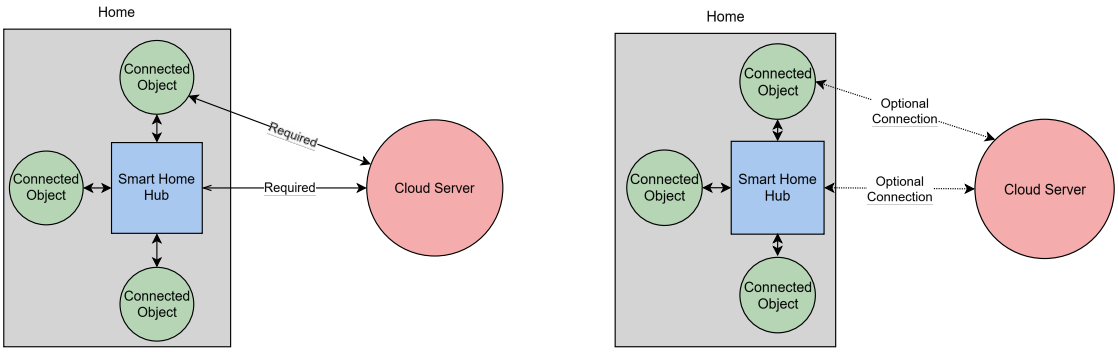
Dependence on external infrastructure. Cloud-based systems introduce a strong dependency on the availability and reliability of external services. If a cloud service becomes unavailable due to technical failure, network issues, or a vendor discontinuing support the associated devices may partially or completely stop functioning. Users do not truly own the devices they have purchased.

Latency and inefficiency. Cloud communication introduces unnecessary latency for interactions that could be handled locally. Unlocking a door from a nearby smartphone, for instance, may still require a round-trip through a remote server even when both devices are on the same local network.

These limitations motivate the need for approaches that reduce reliance on centralized infrastructures while preserving what users actually care about: functionality, reliability, and control over their own home and data.

1.2 From Cloud-First to Local-First Architectures

To address these issues, *local-first* approaches have emerged. Rather than routing all interactions through remote servers, these systems keep data processing and storage within the home by default, only reaching external services when genuinely necessary (see Figure 1.2). The goal is not to eliminate the cloud entirely, but to treat local operation as the norm and cloud access as a deliberate, explicit exception.



(a) Cloud-centric architecture (b) Local-first architecture

Figure 1.2: Comparison of cloud-centric and local-first architectures for smart homes

It is worth distinguishing between *self-hosted* and *local-first* systems. Self-hosting means deploying services on personal hardware (e.g., a Raspberry Pi), while local-first emphasizes that data processing should happen locally *by default*. In practice, local-first systems may still interact with the cloud when genuinely necessary for software updates, remote access, or specific external services. The goal is not to eliminate cloud usage entirely, but to make it the exception rather than the rule, and to make it explicit when it happens.

Platforms such as Home Assistant or OpenHAB [4], [5] are well-known examples of this approach. They act as central hubs within the home, integrating various devices and enabling local automation across Wi-Fi, Bluetooth, Zigbee, and other protocols. These platforms significantly reduce unnecessary data exposure compared to cloud-first alternatives.

However, while local-first platforms improve control and reduce dependence on external infrastructure, they still rely on a coarse-grained model for deciding what applications and services are allowed to do.

1.3 Limitations of Binary Access Control

Existing smart home platforms typically rely on *binary* access control: a device or service is either allowed or denied access to a resource [6]. An application either has access to the camera, or it does not. A service either reaches the internet, or it is blocked.

This model is simple, but it fails to capture the nuance of real-world privacy requirements. Consider the doorbell example again: as illustrated on Figure 1.3, the user may want a third-party application to receive the camera feed, but only a blurred version of it, and only when the doorbell is pressed, not continuously. Binary access control cannot express this. Users end up forced to choose between granting full access and granting none at all.

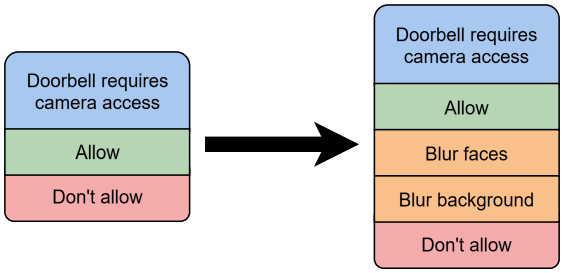


Figure 1.3: Example of binary access control (left) vs. fine-grained access control with PETs (right)

More nuanced policies allowing access only under specific conditions, sharing only transformed or aggregated data, restricting what kind of information an application can extract are simply not expressible in existing systems. Despite growing interest in privacy-aware smart home solutions, current platforms remain largely limited to this coarse-grained model.

1.4 Towards Fine-Grained Privacy Control with PETs

The binary access control model described above creates a fundamental tension in smart home design. Users want applications that are useful, and many useful applications require access to sensitive data. Forcing a choice between full access and no access at all means that, in practice, users tend to grant full access and the privacy constraint is bypassed rather than resolved. What is missing is a mechanism that allows an application to receive data that is useful for its purpose while ensuring that it cannot observe more than what it strictly needs. This is precisely what Privacy-Enhancing Technologies (PETs) are designed to provide.

Definition. The term Privacy-Enhancing Technology covers a wide range of mechanisms: cryptographic techniques such as homomorphic encryption and secure multi-party computation, statistical tools such as differential privacy or anonymization and pseudonymization methods. What these approaches share is the goal of limiting what a data consumer can observe, rather than simply deciding whether it is allowed to observe anything at all [7]. For the purposes of this thesis, we adopt a more specific definition focused on the smart home data-sharing problem : a PET is a software component that intercepts a data stream between a source and an application, applies a transformation locally, and forwards only a modified version to its destination. The transformation can take many forms: blurring or masking regions of a video frame, aggregating fine-grained sensor readings into coarser summaries, filtering specific fields from a structured data object, or scrambling audio signals. The key property is that the downstream application receives only the transformed output; it never has access to the original raw data.

A motivating example. Return to the doorbell scenario of this chapter's opening. A useful response to a doorbell event is for the smart home to send a notification containing a short video clip, so that the user can check remotely who is at the entrance. The application needs some visual information to be useful. But it does not need the full, unprocessed camera stream: it does not need the ability to recognize individual faces, observe background details, or retain a high-resolution recording for later analysis. With a PET integrated into the data path, HubOS can route the raw camera feed through a face-blurring module before it reaches the notification application, as illustrated in Figure 1.4. The application still receives a video clip and can fulfill its purpose. The faces of visitors, neighbours, and passers-by are obscured before any data is transmitted. The raw footage never leaves the hub. This is qualitatively different from binary access control. A rule that simply permits the application to access the camera after a doorbell event does restrict the timing of access, but leaves the content of the stream entirely uncontrolled. The protection is meaningful only if the platform enforces that the raw stream is never made accessible to the application directly, and that the PET transformation is applied regardless of what the application requests. Having a PET running somewhere in the system is not enough; it must be integrated into the permission model itself.

The integration problem. Despite their potential, PETs are rarely integrated into existing smart home platforms in a coherent way. They are typically implemented as standalone tools, disconnected from the permission model of the platform, and difficult for non-expert users to configure correctly [8]–[10]. There is no general mechanism to express, within a single rule, both the conditions under

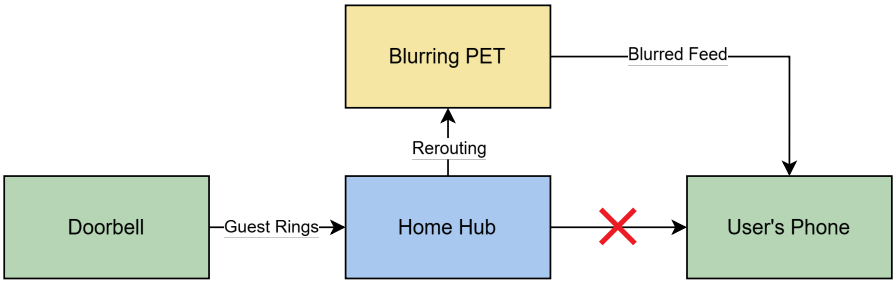


Figure 1.4: Example of PET integration and consequences for data flow and privacy guarantees

which data should be shared and the transformation that must be applied before sharing takes place. The root cause is structural. Existing smart home platforms model data access as a binary permission: an application either has access to a resource or it does not. PETs require the platform to reason about data paths: which module processes which stream, in what order, and under what conditions. Expressing this within a rule-based permission model requires an extension to the model itself, not just a new component bolted on alongside it.

What this thesis proposes. Our approach addresses this gap by treating PETs as first-class components of the smart home framework, modular, reusable, and tightly integrated into the rule model that governs data flows. We extend the TABAC model of HubOS[1] so that a rule can express not only *when* a permission is granted, but also *how* the data must be routed before it reaches its target. A rule can declare that a specific PET module must sit between a raw data source and the application, that the raw source must not be directly accessible to the application, and that the transformation may itself depend on runtime conditions such as the time of day or the active context. The result is a framework in which privacy is enforced not through binary allow-or-deny decisions alone, but through controlled data routing. The doorbell application receives blurred video. A monitoring service receives only high-level detection events, never the raw indoor video stream. A voice-driven service receives a scrambled audio signal rather than a clean copy of the speaker's voice. In each case, the application retains the functionality it needs, and the platform enforces a privacy guarantee that could not be expressed in any existing rule or access control model.

[1]TABAC (Trigger-Action-Based Access Control) is HubOS's permission model, in which access rights are granted dynamically when a trigger fires and its conditions are met. See Sections 2.4 and 4.2 for details.

1.5 Scope and Contributions

Scope. This work focuses on the design and prototyping of a framework that extends HubOS to support fine-grained, transformative control of data flows. The following are explicitly outside the scope of this work: resistance to a compromised hub, OS-level security hardening, physical access control, and network-level intrusion detection. We build a privacy-by-design framework under the assumption that the underlying platform is trusted. The isolation mechanisms our prototype provides are described in detail in later chapters, but we do not claim to offer a fully hardened system. That is a separate and complementary problem.

Contributions. The main contributions of this work are:

- **A conceptual model for non-binary, fine-grained control of data flows in smart homes.** Existing platforms rely on binary allow-or-deny access control, which cannot express policies such as *"share only a blurred version of the video stream, and only when the doorbell is pressed"*. Our model introduces data-flow routing and context-aware rules as first-class primitives, making such policies directly expressible. This closes a fundamental gap between the privacy requirements users actually have and what current rule models can enforce.
- **An integration of Privacy-Enhancing Technologies as first-class components of the permission model.** Rather than treating PETs as standalone tools bolted alongside the platform, we extend HubOS's TABAC model so that a rule can declare both *when* a permission is granted and *how* the data must be transformed before it reaches its target. The raw source is made inaccessible to the application as a consequence of rule activation, not as a separate configuration step. This tight coupling between the permission model and the data-transformation pipeline is what makes the privacy guarantee enforceable and not merely advisory.
- **A prototype implementation validated through representative use cases.** We implement the extended model in a working prototype and demonstrate it on three concrete use cases: a doorbell notification application receiving only blurred video, a living-room monitoring service receiving high-level presence events instead of a raw video feed, and a voice-driven service receiving scrambled audio. These use cases exercise both supported PET integration patterns and show that meaningful privacy properties can be achieved without breaking the application's intended functionality.
- **An empirical evaluation of correctness, isolation, expressiveness, and performance (Chapter 7 and Section 8.7).** We evaluate the

prototype against three criteria: whether the permission rewriting correctly enforces that a target module cannot reach the raw data source once a PET rule is active, whether isolation holds at the network level across all use cases, and whether the rule model is expressive enough to capture real privacy requirements. We also report CPU and memory benchmarks characterising the overhead introduced by PET containers.

Chapter 2

State of the Art

In this section, we review the main families of approaches related to smart home systems, access control, and privacy-preserving computation. The goal is not only to present existing solutions, but also to highlight the limitations that motivate our work. More specifically, we show that current systems provide useful building blocks, such as rule-based automation systems, local hubs, context-aware access control, and privacy-enhancing technologies, but that these elements remain insufficiently integrated into a unified privacy-oriented model.

2.1 Cloud-Centric Smart Home Architectures

Modern smart home systems are mainly dependent on an internet connection. Some connected objects communicate through a central hub, which is responsible for handling the physical and data layers using protocols such as Zigbee, LoRa, or Bluetooth. Other devices connect directly to the internet without going through a hub. In both cases, the overall architecture follows a hub-and-spoke model, where all devices ultimately depend on cloud services. Zavalyshyn et al. describe this model as the dominant architecture in current smart homes and note that it gives platform providers, device manufacturers, and application developers broad access to user data [11].

This approach introduces several constraints. First, there is a strong dependence on internet connectivity. Many devices stop functioning, partially or completely, when the connection is unstable or unavailable. Even simple local interactions can require communication with remote servers, which introduces unnecessary latency. In practice, a user may need to communicate with a server located hundreds of kilometers away to control a device in the same room.

The user is also often locked into the manufacturer's ecosystem and cannot easily extend the capabilities of the system without relying on the manufacturer's

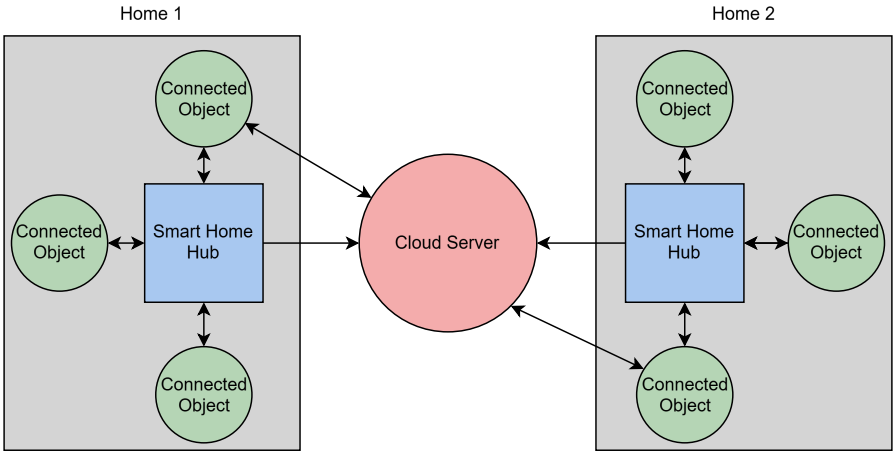


Figure 2.1: Hub-and-spoke system in modern smart homes, with data aggregation in a single location

devices or approval. For example, some smart devices are tightly coupled with a specific ecosystem, such as Apple smart speakers, which can only be fully configured or controlled using another Apple device. This creates a form of vendor lock-in that further reduces the user's control over their own smart home environment.

Data from multiple smart homes is typically transmitted to cloud servers. As a result, manufacturers collect and process large volumes of data originating from different households. This centralization of data introduces significant privacy concerns, as sensitive information about users and their environments is aggregated in a single location (see Figure 2.1). Prior work has further shown that many commercial ecosystems expose users to opaque data flows toward vendors and third-party services, while offering little control beyond a binary choice of connecting or disconnecting devices [11].

Another important limitation is the dependence on the availability and reliability of cloud services. These services are maintained by device manufacturers, and users have little control over them. If a manufacturer decides to stop supporting a device, the user may lose access to some or all functionalities. Even when support is maintained, incidents such as server outages or misconfigurations can disrupt the system. Additionally, users must trust manufacturers to properly secure their infrastructure and handle sensitive data.

Overall, this architecture reduces user control over smart home devices. The system depends not only on the local setup but also on external infrastructure and the decisions of manufacturers. This lack of control is a major limitation when dealing with privacy-sensitive data.

2.2 Local Smart Home Platforms

An alternative to cloud-centric ecosystems is the use of local smart home platforms such as *Home Assistant* and *OpenHAB* [4], [5]. These platforms allow users to manage devices locally, define automation rules, and combine components from different manufacturers in a more open and extensible environment.

This is not only a practical response to the limitations of cloud dependence, but part of a broader design movement around data ownership and local control. Kleppmann et al. argue that local-first systems should prioritize local control and ownership of data while still enabling synchronization and distributed collaboration when needed [12]. Applied to smart homes, this means that data should remain on the hub by default, cloud services should be optional rather than central, and users should retain meaningful control over their own infrastructure (see Figure 2.2). Recent work confirms that the smart home community is gradually moving toward more local processing and edge-based designs [11].

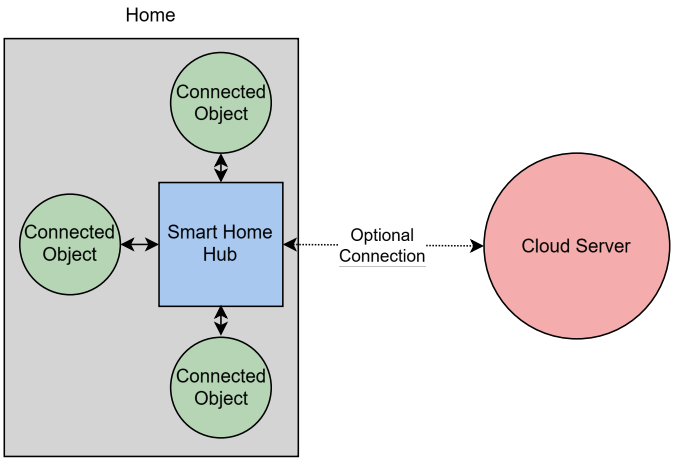


Figure 2.2: Local-First smart home system

Research systems have pushed this idea further. For example, *dSpace* introduces composable abstractions for smart spaces and demonstrates how modular smart home applications can be constructed from reusable components [13]. Such work is relevant because it treats smart home applications as structured compositions rather than monolithic automations, an idea aligned with our treatment of privacy-aware data flows. HubOS applies this local-first perspective directly to smart homes, supporting local processing and minimizing data propagation to remote services [1].

However, local platforms still have important limitations. They primarily focus on orchestration and automation rather than privacy-aware data handling. In addition, common open-source smart home systems do not provide proper isolation

between applications: modules often run under the permissions of a single local user and therefore lack fine-grained restrictions over networking or device access [1]. As a result, while these systems reduce cloud dependence, they do not by themselves solve the problem of privacy-preserving data sharing between components.

2.3 Privacy-Aware Smart Hubs

A more directly relevant line of work consists of privacy-aware smart hubs. Instead of treating the hub only as a communication relay, these systems use it as a trusted control point where privacy policies can be enforced locally. This idea is central in the literature surveyed by Zavalyshyn et al., who show that hub-based approaches can reduce data exposure through local storage, local computation, flow filtering, and controlled communication with the cloud [11].

An early and important example is *HomePad*, a privacy-aware smart hub that gives users more control over how applications access and process sensitive data [14]. HomePad models applications as directed graphs of processing elements and uses Prolog-based reasoning to verify whether a given application violates user-defined privacy policies. This is a strong contribution because it moves beyond simple device permissions and introduces explicit reasoning over data flows. HomePad also demonstrates that privacy requirements may involve transformations of the data itself, for example blurring images or rate-limiting transmissions.

More recently, *HubOS* proposes a local-first operating system for smart home hubs that supports isolated application modules and context-sensitive access control [1]. HubOS is particularly relevant for this thesis because it combines local execution, application isolation, and a rule-based access control model called TABAC (introduced in Section 2.4). It shows that modern smart home applications can be decomposed into modules, run locally on a hub, and be granted access to resources only under specific runtime conditions.

Another particularly relevant contribution is *Peekaboo*, which integrates privacy-preserving operators directly into automation pipelines and allows users to inspect and constrain how data is transformed before being shared [15]. Peekaboo is especially important because it moves closer to treating privacy-preserving transformations as part of rule execution itself rather than as an external add-on.

Complementary work such as *PFirewall* approaches the problem from a data-flow control perspective. Instead of explicitly transforming data, PFirewall analyzes automation semantics to infer what disclosures are necessary and blocks all other traffic [16]. This demonstrates another promising direction in which privacy protection is embedded into the behavior of the automation system itself.

These privacy-aware hubs represent a major step forward compared to classical smart home platforms. However, despite significant progress, existing approaches

remain specialized and do not yet provide a general and reusable framework in which access control, rule execution, and privacy-preserving transformations are jointly expressed.

2.4 Access Control Models

Traditional access control models such as Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC) are widely used in distributed systems to regulate access to resources [17], [18]. RBAC assigns permissions based on roles, while ABAC bases decisions on attributes of subjects, resources, and context.

Beyond classical RBAC and ABAC, more expressive models have been proposed for smart home environments. *HyBAC*, for instance, combines the structured role management of RBAC with the contextual expressiveness of ABAC, aiming to better handle the dynamic nature of IoT settings [19]. These hybrid approaches improve on static permission models, but they still treat access as a binary decision: a subject either gets access to a resource or it does not. Contextual policies such as "allow the camera only when the doorbell rings" are difficult to express, and none of these models support sharing only a transformed version of the data.

HubOS addresses the context problem with TABAC (Trigger-Action-Based Access Control), which makes permissions event-driven [1]. In TABAC, a permission is granted dynamically when a specific trigger fires and its conditions are satisfied, then automatically revoked after a set period. For instance, a TABAC rule might grant an external application temporary access to a camera stream whenever the doorbell rings, and revoke it after a set period. This aligns access decisions with the runtime state of the smart home rather than with a static configuration set at install time.

However, TABAC still assumes that data is shared in its original form. A rule can decide when a module may access a camera stream, but it cannot express that the stream should be blurred before it reaches the application. The behavioral automation layer of smart home platforms, rule-based systems, operates alongside these access control models. As we discuss next, it faces the same fundamental limitation.

2.5 Rule-Based Systems

Most smart home rule languages adopt a trigger-condition-action (TCA) paradigm: a rule fires when a trigger occurs and its conditions are satisfied, then executes a predefined action [20]–[22]. In platforms such as OpenHAB or Home Assistant [4], [5], triggers can be temporal events, device state changes, or sensor readings, and

actions are commands sent to actuators. This model improves on static access control by making behavior conditional on the current context.

However, TCA rules have an important structural limitation: actions are predefined, device-level commands (switching a device on, setting a temperature) rather than data-transforming computations [20], [22]. A rule can decide *when* something happens, but not *how* the data involved is handled. This makes it difficult to express behaviors such as "share the camera feed when the doorbell rings, but blur faces before transmitting" or "forward only a high-level detection event rather than the raw video stream." The rule system has no mechanism to sit between the data source and the application and modify what passes through.

This limitation is compounded by a structural gap between rule systems and access control models. Context-sensitive access control models can express fine-grained, context-dependent permissions over who may access which data [23], [24], but they operate as separate policy engines, decoupled from end-user rule scripting. Users therefore cannot express within a single rule both the conditions under which data is shared and the transformation that must be applied before sharing takes place.

Surveys and frameworks on user-centric privacy in smart homes confirm this diagnosis: existing solutions still lack integrated, context-aware, privacy-preserving data-sharing mechanisms in mainstream platforms [25]. While one layer defines permissions and the other defines behavior, neither provides a unified way to express fine-grained, context-aware, and privacy-preserving data sharing. Privacy-Enhancing Technologies enable a different approach to this problem: rather than deciding when or whether data is shared, they act directly on the data itself, transforming it before it reaches its destination.

2.6 Privacy-Enhancing Technologies

As defined in Section 1.4, PETs are mechanisms that reduce the exposure of personal information while preserving some utility of the data [26], [27]. They include a wide range of approaches that share the goal of limiting what a data consumer can observe, rather than simply deciding whether it is allowed to observe anything at all. For the purposes of this thesis, we adopt the narrower definition focused on the smart home data-sharing problem: a PET is a software component that intercepts a data stream, applies a transformation locally, and forwards only the modified version to its destination, so that the privacy protection comes not from restricting when data is shared, but from transforming what is shared.

Concrete proposals in the smart home literature illustrate several of these forms. *PrivHome* combines privacy-preserving authenticated communication and encrypted data handling for smart home environments [28]. Other work applies

local differential privacy to smart home telemetry to protect individual readings while preserving aggregate utility [29]. *Peekaboo* goes further by integrating privacy-preserving operators directly into automation pipelines, treating transformations as part of rule execution rather than as external tools [15]. The survey by Zavalyshyn et al. also documents several hub-based PETs relying on local filtering and hybrid trust models [11].

However, in most existing platforms PETs remain disconnected from the rule and permission models. A user can define when an application is allowed to access a data source, and separately deploy a transformation component, but no standard mechanism ties these two concerns together in a single enforceable policy [10]. As a result, PETs are often underused in practice despite their clear potential [30].

A less technical but equally important challenge is that of user understanding and trust. For a PET to provide meaningful protection, users must be able to tell that it is active and correctly applied. Research on smart home privacy shows that users generally have limited visibility into how their data flows through the system and which services receive what [25]. When a transformation is happening, it is largely invisible: users cannot easily verify that blurring was applied, that the raw stream was not also forwarded elsewhere, or that the PET was started at all. This opacity matters because a PET that users cannot observe provides weaker guarantees in practice, even if it is technically correct. It also suggests that PET integration needs to be accompanied by some form of transparency, making the data path visible and understandable rather than hidden inside the system. These challenges (technical integration, enforcement, and user transparency) are part of the broader fragmentation that the following section examines.

2.7 Limitations of Existing Approaches

Taken together, the literature reveals substantial progress, but also fragmentation. Some systems focus on local execution and isolation, others on context-aware authorization, and others on privacy-preserving transformations or data minimization.

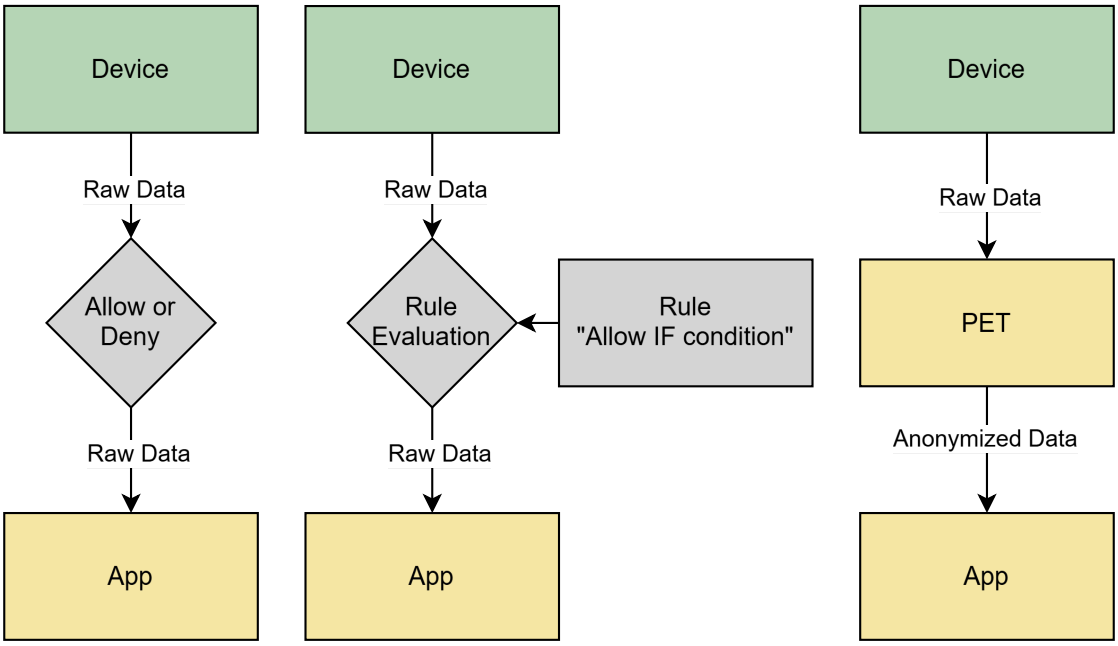
Recent systems such as HomePad, Peekaboo, and PFirewall begin to connect some of these dimensions, for example by reasoning over data flows, embedding privacy operators into automation pipelines, or restricting unnecessary disclosures based on rule semantics. However, these approaches remain specialized and do not provide a unified and reusable model in which access control, rule execution, and privacy-enhancing transformations are expressed within the same policy framework.

In particular, existing systems still provide limited support for policies that are simultaneously:

- fine-grained at the level of individual data flows,

- context-aware at runtime,
- capable of transforming data before disclosure,
- and directly embedded into the smart home automation and access-control model.

This gap, illustrated in Figure 2.3, motivates the approach proposed in this thesis. Our goal is precisely to connect these currently separate dimensions by integrating PETs into a rule-based and context-aware access control model, so that privacy is enforced not only through binary authorization decisions, but also through controlled transformation and abstraction of the data itself.



(a) Binary model; "on or off" access control without data transformation

(b) Context-aware model; rules can express when data is shared but not how it is transformed

(c) Privacy-aware model; PETs can transform data but are not integrated into the rule and access control system

Figure 2.3: Gap analysis of existing approaches: no unified model that integrates all these dimensions together

This positioning leads naturally to the design principles introduced in the next chapter. If current systems are limited because they separate access control, rule execution, and privacy-enhancing transformations, then the solution must treat

data flows as first-class objects and allow privacy control to be expressed directly in the execution model itself.

Chapter 3

Vision and Design Principles

This chapter presents the conceptual foundations of our approach to privacy control in smart home environments. Rather than relying on traditional binary access control mechanisms, we propose a model in which data flows are dynamically mediated through context-aware rules and Privacy-Enhancing Technologies (PETs). We begin by defining who participates in the system and what each party is assumed to trust, then introduce the design principles that follow from those assumptions.

3.1 Actors and Trust Model

Before describing the design principles, it is worth being explicit about who participates in the system and what assumptions we make about each party. Our goal is not to defend against a compromised hub or software that exploits OS-level vulnerabilities. The problem we address is different: a legitimate application that receives more data than it needs.

Actors. Five types of actors are involved :

- *User*: the person who owns and controls the smart home.
- *HubOS platform*: the local software that runs on the hub, enforces rules, and mediates all data flows between components.
- *Application modules*: third-party services installed on the hub, for example a doorbell notification service or a monitoring dashboard. They provide useful functionality but are developed independently and may request broader data access than their purpose requires.

- *PET modules*: transformation components that sit between a raw data source and an application. They process data locally and forward only a modified version to its destination.
- *External services*: remote APIs or cloud endpoints that are outside the user's direct control, and that some applications may communicate with.

Trust relationships. The user trusts HubOS. The user does not necessarily trust application modules: even a well-intentioned application may request access to more data than it strictly needs, and the user has no direct way to verify what the application does with that data. HubOS therefore treats all application modules as potentially over-privileged and enforces data flow restrictions on their behalf, rather than relying on the application to self-limit. PET modules are trusted to perform the transformation they declare, but they are not trusted with the raw data beyond what their declared function requires. HubOS enforces this boundary by granting the PET only the access it needs to perform its transformation, and granting the downstream application only the transformed output.

Threat model. The threat this thesis addresses is subtle but common: a legitimate application that receives more data than necessary. An application that correctly performs its stated function may still collect more than needed, retain data longer than expected, or enable re-identification through combination with other sources. These harms do not require any exploit or malicious intent. They follow directly from a permission model that grants access in binary terms, without constraining what the application can observe. Our framework addresses this by interposing a PET in the data path, so that the application receives a transformed version of the data rather than the raw stream, regardless of what the application requests.

PET providers and auditability. PET modules can be developed by any party: device manufacturers, independent developers, or open-source contributors. Because a PET sits in the data path and has access to the raw stream, the question of who develops it and whether its behavior can be verified matters. The ecosystem proposed in this work, described in Section 4.5, includes a community store through which PET modules and rule files can be shared and reviewed. This makes the trust relationship explicit: users can inspect which PET is active, what transformation it applies, and whether the rule prevents the application from accessing the raw source directly. Community auditability is one mechanism for building this trust; it does not eliminate all risk, but it makes the data path observable rather than opaque.

3.2 Local-First Privacy Control

We aim to introduce a privacy-oriented smart home environment. To achieve this, it is necessary to restrict the exposure of data to external services. These services reduce user control over both the smart home system and the data generated by it.

Local processing allows the user to regain control over their smart home and its data. A local-first approach means that data does not leave the local environment by default and is processed directly on the user's hub or devices. This approach comes with multiple advantages. It reduces dependency on external infrastructure and avoids vendor lock-in, enabling a more open ecosystem. The user remains in control of their data and has better control over their devices, without relying entirely on the manufacturer.

3.3 Data Flow as a First-Class Concept

Simply managing privacy through basic permission systems is limiting and does not allow for fine-grained control. Traditional models focus on whether a service can access another service, as shown in Figure 3.1a, but this does not capture how data is shared or used.

To address this limitation, we propose to manage privacy at the level of data flows. Instead of controlling connections between services, we control how data moves between them. A data flow refers to the movement of data between different components of a system, from its source (e.g., a sensor) to one or more destinations (e.g., applications or services), as shown in Figure 3.1b. It describes not only where the data originates and where it is sent, but also how it is processed, transformed, or filtered along the way.

In the context of smart home systems, data flows represent how information such as sensor readings or video streams is shared and potentially modified before being consumed by other components. By focusing on data flows, we are no longer limited to allowing or denying access between services. Instead, we can control each flow individually, enabling more precise and flexible privacy policies.

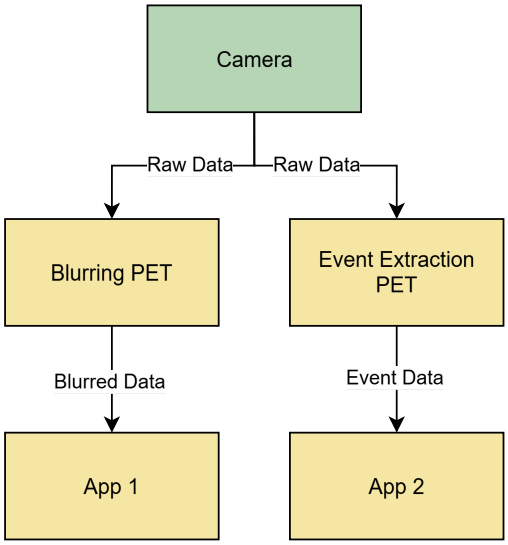
3.4 Context-Aware Policies

Controlling data flows is not sufficient on its own. It is also important that this control depends on the context in which the system operates. Privacy requirements are rarely static and often depend on time, events, or environmental conditions, even more so in a smart home environment.

In our model, data flow control is therefore context-aware. This means that permissions can change dynamically depending on the situation. For example, one

	Camera	Thermo
App 1	YES	YES
App 2	NO	YES
App 3	YES	NO

(a) Static model with binary permissions



(b) Data flow model

Figure 3.1: Comparison of a traditional binary permission model and a data flow model

service may be allowed to send video data to another service at a given moment, but not telemetry data. At another time, the opposite may be true. Similarly, access may depend on events such as a doorbell being triggered.

Context-aware policies allow the system to adapt its behavior dynamically and provide a more expressive and realistic way to manage privacy. They enable fine-grained control over what data is shared, when it is shared, and under which conditions.

3.5 PETs as Modular Components

Having context-aware data flow control allows us to introduce PETs as intermediate components in the system. As defined in Section 1.4, a PET intercepts a data stream, applies a transformation locally, and forwards only a modified version to its destination (Figure 3.2). What matters here is their role as *modular, reusable* components: the same face-blurring module can sit in the doorbell data path, the living-room monitoring path, or any other video flow where the user chooses to apply it. Different PETs can be swapped in for the same flow without changing the rule structure, and new PETs can be contributed independently of the applications that use them.

Combined with fine-grained, context-aware data flow control, PETs enable a

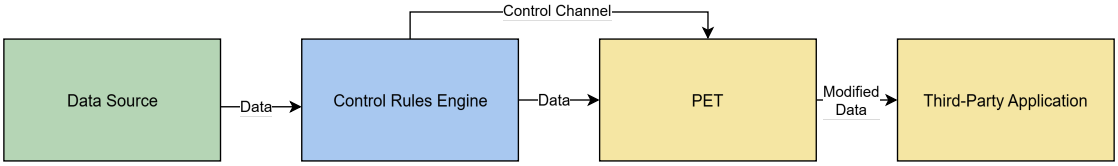


Figure 3.2: Data flow through a PET

non-binary approach to privacy. Instead of simply granting or denying access, the system can transform or abstract data before sharing it, reducing the exposure of sensitive information while preserving functionality.

This concept of PETs as modular components, as well as the concept of context-aware policies, form the core of our approach to privacy control in smart home environments. These concepts are introduced here as design principles only; chapter 4 presents how we integrated them into a concrete model, and introduces the ecosystem in which they can be developed and shared. Chapter 5 then describes the system architecture and its implementation in the HubOS platform, and chapter 7 evaluates its correctness and expressiveness.

Chapter 4

Proposed Model

This chapter presents the model we propose to extend HubOS with fine-grained, privacy-aware data flow control. We begin by describing the conceptual shift from binary access control to transformative control enabled by PETs, then introduce the TABAC rule structure and the extension that allows PET behaviour to be expressed within it. We then describe the programming model through which developers write these rules, the access control mechanisms that enforce them at runtime, and the ecosystem in which applications and rules are distributed.

4.1 From Binary to Transformative Control

By introducing PETs, we extend access control beyond binary decisions and enable the system to enforce more nuanced privacy policies.

In this model, access is no longer defined only by permissions between services, but also by how the data is handled during its transmission. For example, a service may be allowed to access a data flow, but only after it has been filtered, anonymized, or aggregated.

This shift from binary to transformative control allows for more flexible and realistic privacy management. It enables the system to preserve functionality while reducing the exposure of sensitive information, and better reflects the complexity of real-world smart home scenarios.

4.2 Extension of TABAC Rules

HubOS governs data access through TABAC, a rule-based model in which each rule has three main elements. A **trigger** defines the event that activates the rule, a doorbell press, a sensor update, or a user interaction. A set of **conditions** refines when the rule should actually fire. For example, only during certain hours or

when a specific device is in a given state. A set of **actions** defines the temporary permissions granted once the trigger fires and all conditions are satisfied, typically access to a device, a data stream, or an external service, scoped to a defined period after which it is automatically revoked.

This structure is context-aware and enforces least privilege. Its limitation, however, is that it treats access as binary: it controls *when* a module may access a resource, but not *how* the data is handled once access is granted. A rule cannot express "share the video, but only a blurred version outside these hours." We extend TABAC to close this gap.

4.2.1 PET Rules Execution Model

To address these limitations, we introduce PET rules as an extension of TABAC rules. A PET rule is attached to a regular TABAC rule and evaluated as part of its execution: when the rule fires, the system performs the standard trigger-condition-action processing, then evaluates any attached PET rules. Those PET are therefore tightly integrated into the rule execution pipeline, and act as context-aware runtime handlers that can modify the data flow between the source and the target module.

Each PET rule carries its own conditions, evaluated independently from those of the parent rule. If the conditions are met, the PET is activated and the transformation is applied before the data reaches its destination. If they are not, the route still passes through the PET but the transformation is disabled, so the target module always connects to the same output address regardless of the current condition state. This process is illustrated in Figure 4.1.

This differs from traditional rule-based systems in an important way: instead of only deciding whether an action should be executed, the system also controls how the data involved in that action is processed. PETs act as intermediate steps in the execution pipeline, and their behaviour can change at runtime without changing the permissions the target module holds.

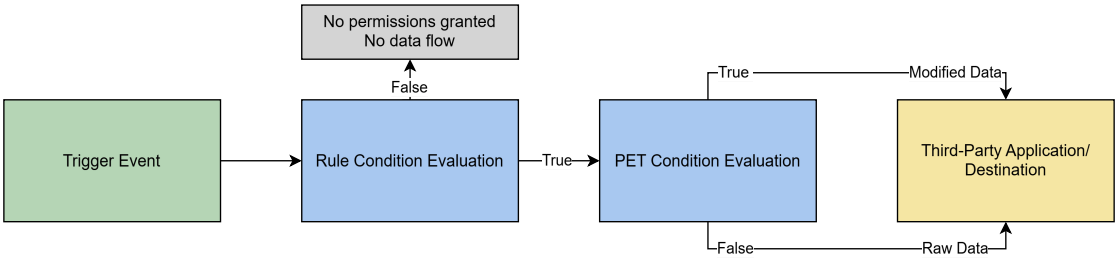


Figure 4.1: PET rule execution

4.3 Programming Model

Rules are written as JSON objects and stored in a file called `rules.json`, which is part of each application package. This section describes the format, focusing on the PET extension. The execution behaviour that follows from these rules is described in the previous section; what matters here is what a developer actually writes.

4.3.1 Base TABAC rule

A standard TABAC rule has four top-level fields: `name`, `when`, `condition`, and `then`.

```
{
  "name": "ring_button_call_ring_handler_module",
  "when": {
    "event": "device.ringSwitchItem",
    "context": "updated",
    "value": "ON"
  },
  "condition": [],
  "then": [
    {
      "access": "NetworkClient",
      "type": "service",
      "context": {
        "period": "600",
        "host": ["telegramServer", "CameraUrlBlurred"],
        "concern": "ring_handler"
      }
    }
  ]
}
```

The `when` block defines the trigger: which event to listen for, and what value it must carry. The `condition` array can hold additional checks that must pass before the rule fires. The `then` array lists the permissions to grant when it does: here, the `ring_handler` module receives temporary network access to the Telegram server and to a camera stream, valid for 600 seconds.

4.3.2 Adding a PET rule

The PET extension adds an optional `pet` array at the top level. Its presence does not change how the TABAC part of the rule is evaluated. It only affects which permissions the target module actually receives.

```
"pet": [
  {
    "name": "blur_camera_stream_outside_allowed_hours",
    "type": "stream_transform",
    "mode": "transparent",
    "pet_module": "stream_blur",
    "source": "CameraUrl",
    "output": "CameraUrlBlurred",
    "target": "ring_handler",
    "condition": [
      {
        "name": "outside_school_hours",
        "if": {
          "event": "system.time",
          "context": "not between",
          "value": ["08:00", "16:00"]
        }
      }
    ],
    "mqtt": {
      "command_topic": "pet/stream_blur/command",
      "status_topic": "pet/stream_blur/status"
    },
    "config": {
      "input_url": "https://camera.local/mjpg/video.mjpg",
      "blur_type": "faces",
      "duration_seconds": 300,
      "output_path": "abc123"
    }
  }
]
```

The key fields are:

- `pet_module`: the name of the PET module that will process the data. It must match a module declared in the application configuration.

- `source`: the name of the raw data server, as declared in the system configuration. This is the resource that the target would have received access to under a plain TABAC rule.
- `output`: the name of the server that the PET will expose after transformation. The target module receives access to this instead of the raw source.
- `target`: the module that should receive the transformed data. The Permission Manager uses this to identify which grant to rewrite.
- `condition`: an array of conditions evaluated at runtime. If the conditions are met, the PET is started with the configured transformation. If they are not, the route still goes through the PET, but the transformation is disabled. This means the target always connects to the same address regardless of the current condition state, which avoids the need to change permissions mid-rule.
- `mqtt`: the command and status topics that the PET and the target module can use to exchange control messages. Access to these topics is optional, granted for the duration of the rule, and revoked when it expires. In our examples, MQTT is only used in 6.2.
- `config`: a configuration object forwarded to the PET when it starts. Its content depends on the PET being used. For the video blur PET, it includes the input URL, the blur mode, and how long to run.

The `type` and `mode` fields describe the routing pattern. `stream_transform` with `transparent` mode means the target still receives a stream at a URL, just routed through the PET. The application does not need to know that any transformation is happening.

4.3.3 MQTT-based PET

Not all use cases involve a continuous stream. When the target module only needs derived information from the data rather than the stream itself, the PET communicates through MQTT instead. In this case, the `config` object sets `autostart` to `false`, and the target module is responsible for sending the start command to the PET once it receives its permissions.

```
"pet": [
  {
    "name": "keyword_watch_pet_route",
    "type": "stream_transform",
```

```
    "mode": "transparent",
    "pet_module": "keyword_watch",
    "source": "ConfiguredVideoStream",
    "output": "KeywordWatchEvents",
    "target": "keyword_pet_dashboard",
    "condition": [],
    "mqtt": {
      "command_topic": "pet/keyword_watch/command",
      "status_topic": "pet/keyword_watch/status"
    },
    "config": {
      "autostart": false,
      "duration_seconds": 300,
      "output_path": "keyword-watch"
    }
  }
]
```

Here, the `keyword_watch` PET processes the raw video stream locally and publishes detection events on its status topic. The dashboard module never receives the video at all. It sends a start command with the keywords it is looking for, then waits for detection messages over MQTT. The raw stream permission is held by the PET; the dashboard holds only the MQTT topic grants.

4.3.4 PET conditions

The `condition` array inside a PET rule is evaluated independently from the conditions in the main TABAC rule. A condition specifies an event, a comparison context, and a value:

```
{
  "name": "outside_school_hours",
  "if": {
    "event": "system.time",
    "context": "not between",
    "value": ["08:00", "16:00"]
  }
}
```

When the condition is met, the PET starts with the transformation active. When it is not met, the PET still starts but with transformation disabled, so the

output stream passes the data through unchanged. In the current implementation, only time-based conditions are supported.

An empty `condition` array means the PET is always activated with the transformation enabled whenever the parent rule fires, as in the voice scrambling use case where no time window is needed.

4.4 Access Control Model

Permissions are granted dynamically as part of rule execution. When a rule is triggered, the system determines which services need access to which resources in order to perform the required actions. This includes both the main services involved in the rule and the PET modules that may process the data.

Instead of granting permanent access, permissions are scoped in time and context. They are only valid during the execution of the rule and only for the specific data flows involved. This reduces unnecessary exposure and limits the impact of potential misuse.

Access control is applied at the level of data flows rather than at the level of devices. When a PET rule is present, the Permission Manager intercepts the grant before it is issued. Rather than giving the target module direct access to the raw data source, it withholds that grant and issues two separate ones: the PET module receives access to the raw source, and the target module receives access only to the protected output alias.

In practice, permissions are enforced through MQTT-based communication between components. Services are authorized to access specific network resources, and PET modules are granted the necessary rights to process and transform data. When a rule declares MQTT communication topics for a PET, the Permission Manager grants access to exactly those declared topics and no others, and only for the duration of the rule.

By combining dynamic permissions, data flow control, and PET integration, the access control model moves beyond simple allow/deny decisions. It enables fine-grained, context-aware authorization that aligns with the privacy requirements of smart home environments. The implementation of these mechanisms is described in Chapter 5.

4.5 Ecosystem

We propose an ecosystem where applications, PETs and TABAC rules are made available through a community store (see Figure 4.2).

In this model, PET modules can be distributed and shared via this store. Each

third-party application is composed of two main elements: the application code itself and a TABAC rule file that defines which components need access to which resources, and under what conditions. This TABAC rule file includes both standard TABAC rules and the associated PET rules.

The rule file is initially provided by the application developer, but it is not fixed. It can be replaced or adapted by a community-provided version, which may enforce stricter or more permissive policies depending on user preferences. These alternative rule files are also published and shared through the community store, allowing users to choose the level of control and privacy that best fits their needs.

While this ecosystem enables flexibility and customization, it also introduces important security and trust challenges. Allowing third-party developers to publish applications and rule files creates the risk of malicious or poorly designed components that could request excessive privileges, bypass intended privacy protections, or embed harmful logic. Similarly, community-provided rule files may not always align with a user's privacy expectations. As a result, mechanisms for establishing trust, verifying application behavior, and validating rule correctness become essential in such an open ecosystem. These challenges are discussed in more detail in Chapter 8.

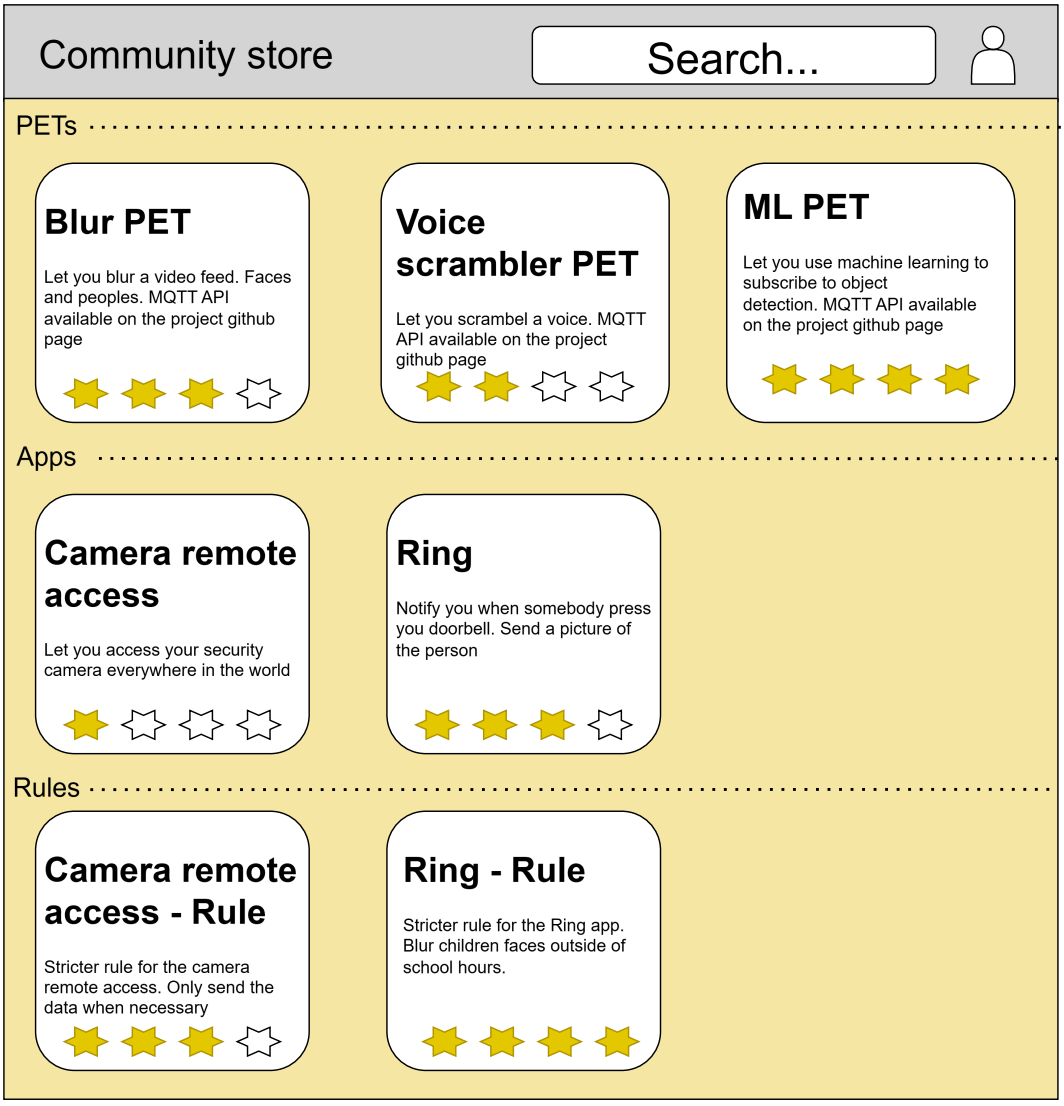


Figure 4.2: Community store

Chapter 5

Implementation

This chapter describes how we turned the proposed model into a working system. The focus of this chapter is: *how* we built the system.

We carried out this work on HubOS, an experimental smart-home platform built on top of OpenHAB that introduces TABAC rules. We forked HubOS and extended it with PET rules, so that the same rule that grants access can also dictate how the data is transformed before it reaches its destination. The extension adds roughly 11,500 lines of code across 86 files under `src/` (close to 20,000 insertions in total once generated benchmark data and plots are included).[1] A central constraint throughout was *backward compatibility*: a rule that does not use the new `pet` field must be parsed, decoded and enforced exactly as it was before, so that the original TABAC behaviour is preserved unchanged.

5.1 Building on HubOS

A recurring question for this work is which guarantees come from HubOS itself and which are our own contribution. The distinction matters because the privacy properties of the PET extension rest on isolation primitives that were introduced by HubOS. We did not re-implement sandboxing, the OpenHAB integration, or the MQTT-based permission machinery; we built the PET layer on top of HubOS and reused them as the trust anchor for our additions.

HubOS provides the *enforcement substrate*, gVisor-isolated [2] per-module containers, a permission-gated egress proxy, the compilation of TABAC rules into native OpenHAB rules, an MQTT broker with per-module identities, and a time-bound permission model. We added the *privacy layer* on top of it. Our additions

[1] The full source is available at https://github.com/riri-314/hubos_2.

[2] gVisor is an open-source container sandbox that provides an additional isolation layer by implementing a kernel running in the user-space. It is written in a memory-safe language (Go)

are the PET rule type and its decoding path, the permission-rewriting engine, an active expiry mechanism that revokes and announces grants as they lapse, a northbound MQTT interface through which applications observe and release their authorisations, the three PET modules with their third party applications. We also implemented a prototype of a web interface that visualises the active rules and grants (Section 8.3.2). The only inherited component we modified rather than simply reused is the container network, which we relaxed from a fully internal bridge to a proxy-gated one (Section 5.3).

5.2 System Architecture

This section describes the components that make up the system (see Figure 5.1) and how they interact at runtime when a rule fires.

5.2.1 System Components

We distinguish two main categories of components: system components and third-party applications.

System components include the central system application (HubOS in our implementation) and the MQTT broker. The system application is responsible for evaluating TABAC rules, orchestrating the execution of actions, and managing authorizations. It has access to all data flows and acts as the central control point of the system. The MQTT broker is used as the main communication layer, allowing components to exchange commands and control messages. While some data flows, such as video streams, may be transmitted using protocols like HTTP or HTTPS, their orchestration is still managed through MQTT.

Third-party applications include both external services and PET modules. These components are considered untrusted by default. They are sandboxed and do not have access to any data or resources unless explicitly authorized by the system. The system application must grant permissions for each data flow or resource they need to access.

PET modules are treated similarly to third-party applications. Even though they are used to enhance privacy, they are not inherently trusted and must also be explicitly authorized. This design choice ensures a consistent and secure model where all external components are controlled in the same way.

This approach enables a privacy-first system while still allowing extensibility. Developers can build new services and PETs, but they must operate within a controlled environment where access is explicitly managed by the system.

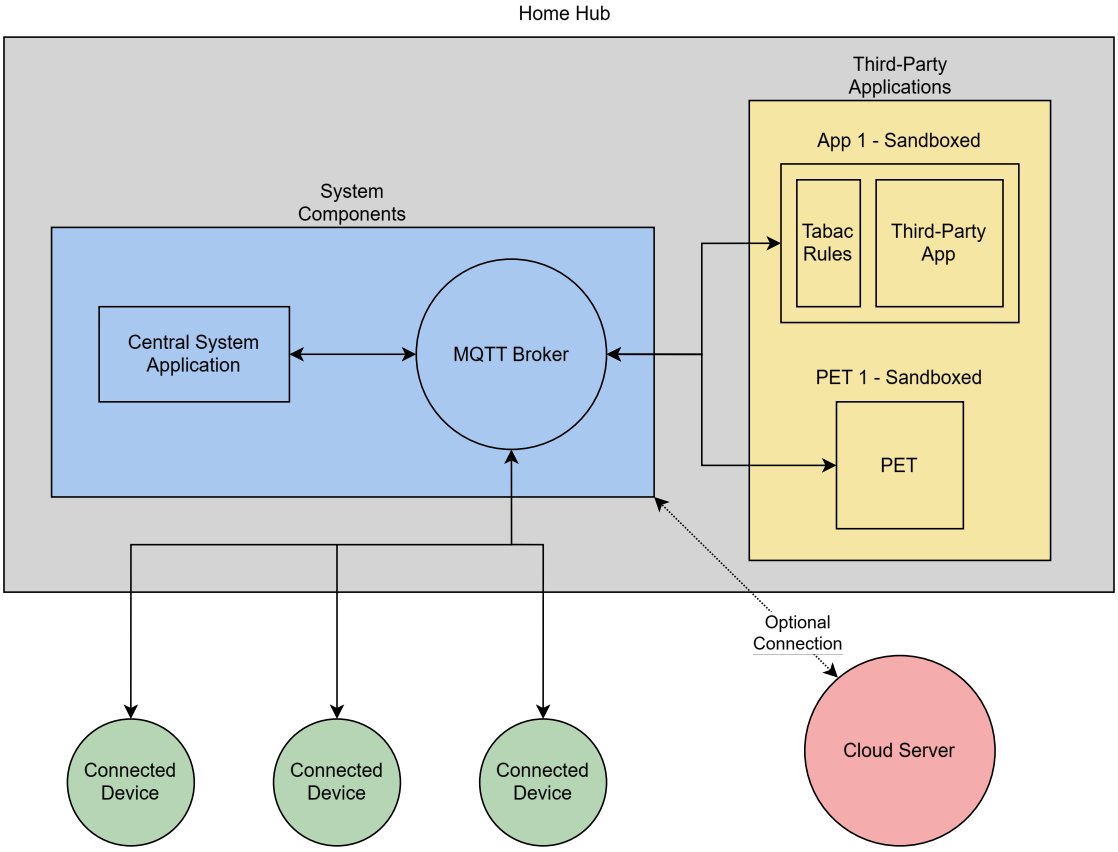


Figure 5.1: System components

5.2.2 Data Flow

This subsection describes how the main components interact when a rule fires. The execution model itself is described in Chapter 4, the focus here is on which component is responsible for each step and how they communicate. Figure 5.2 provides a visual representation of the process.

When an event matching a rule trigger arrives, the TabacManager evaluates the rule and assembles an activation payload. This payload contains the list of permission grants from the `then` block and the PET specifications from the `pet` block, along with the flow values to be forwarded to target modules. The TabacManager sends this payload to the PermissionManager over an internal MQTT topic (`admin/rule-auth`).

The PermissionManager processes the payload in two passes. First, it builds the standard permission grants for each target module. Then, for each PET specification, it rewrites those grants: the raw source is removed from the target's permissions and assigned to the PET module instead, while the target receives

access to the PET's output address. If the PET specifies MQTT topics for control communication, the PermissionManager also grants access to exactly those topics for the duration of the rule.

Once permissions are written, the PermissionManager sends a supervision message to the PET module instructing it to start, and a flow message to the target module carrying the output stream URL and any other values the rule defined.

Permissions are time-bound. The PermissionManager sets a timer for the period declared in the rule. When it expires, all grants are revoked: the target loses access to the PET output, the PET loses access to the raw source, and any MQTT topic grants are removed.

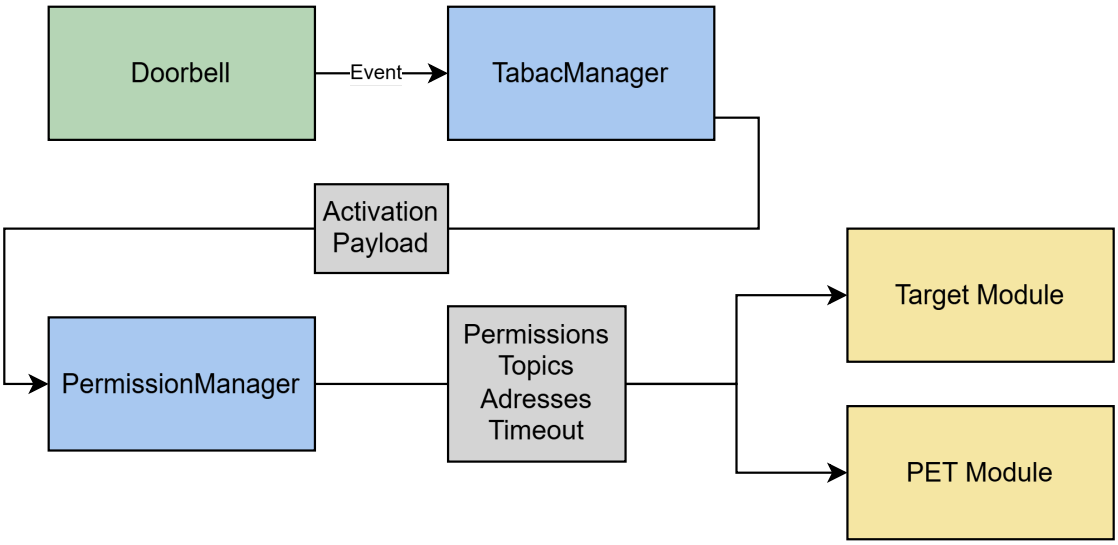


Figure 5.2: Data flow when a rule fires

5.3 The Sandboxing Model

HubOS treats every application and PET as untrusted. A PET receives no special trust: like any other application, it must be granted access to a data source by the PermissionManager before it can read it. This uniform treatment is a deliberate design choice and is discussed at the architectural level in the System Architecture chapter; here we describe how the isolation is actually enforced.

Two layers of isolation. Each module is packaged as its own Docker image and run as a single container, created in `SandboxManager.createContainer()`. Two mechanisms confine it. First, the container uses the gVisor runtime (`Runtime:`

'runsc') rather than the host kernel, so a compromised module is contained by a user-space kernel rather than sharing the host's. Second, and more importantly for data flows, the container has *no direct route to the network*: its `HTTP_PROXY` and `HTTPS_PROXY` environment variables point at a HubOS-controlled forward proxy (`HProxy`, built on the `straightforward` library), and the container is attached only to the internal `hubos_net` bridge. This was inherited from HubOS.

How egress is enforced. On every request and `CONNECT`, the proxy reads the caller's Basic-Auth credentials from the `Proxy-Authorization` header, the username is the module's identifier, and asks the PermissionManager whether that module may reach that host (the `isServerPermitted` check). If no currently valid permission grants that access, the proxy destroys the socket. Because a freshly started module holds no permissions, *all outbound traffic is denied by default*; a host becomes reachable only while a TABAC rule has granted it, and only for the rule's declared period. Access to devices and to MQTT topics is gated the same way, through grants delivered over MQTT and enforced at the broker. We treat PETs as ordinary sandboxed applications under exactly this model.

What we changed, and why. At the fork point the `hubos_net` bridge was declared fully internal (`Internal: true`, IP masquerading disabled), so a container could not route off the host even if it bypassed the proxy environment variables. We relaxed this to a non-internal bridge (`Internal: false`) because PET outputs are served on the host and re-exposed through the HubOS stream API: a target application reaches the transformed stream through a host endpoint, which requires the container to be able to reach host-provided services. The consequence is that the default-deny egress guarantee now rests entirely on the proxy and the permission check rather than on a hard network wall as well. This is a conscious trade-off; the proxy remains the single point at which every outbound connection is authorised.

5.4 Extending the Rule Pipeline

When an application is loaded, HubOS reads its `rules.json` file and runs it through a three-step pipeline before registering anything with the OpenHAB rule engine. This load-time pipeline is where most of the PET parsing logic lives; the runtime sequence that follows when a rule later fires is described in the System Architecture chapter. We kept the existing pipeline structure and extended each step to recognise the optional `pet` field.

Step 1 - extraction. `TabacRules` reads the file, parses the JSON array, and constructs a `TabacRule` object for each entry. If an entry contains a `pet` array,

each element is wrapped in a `TabacPet` object and attached to the rule; entries without a `pet` field default to an empty list and pass through unchanged. This default (`rule['pet'] || []`) is the first of three places where backward compatibility is preserved: a rule with no PET ends up with an empty PET list and is indistinguishable from a pre-extension rule.

Step 2 - linking. `TabacRule.linkEntityReferences()` resolves the symbolic names used in the rule to concrete system identifiers. For standard actions, server names are mapped to host addresses and module names to internal module IDs. For PET rules, `TabacPet.linkEntityReferences()` additionally resolves the `pet_module` and `target` names to module IDs and the `source` and `output` names to host addresses. If any referenced module or server is absent from the system configuration, a `TabacError` is thrown and the rule is not registered. This is intentional: a misconfigured rule fails loudly at load time rather than silently at runtime.

Step 3 - decoding. `TabacRule.decode()` translates the rule into the action format accepted by the OpenHAB rule engine, and this is where PET rules diverge from plain TABAC rules.

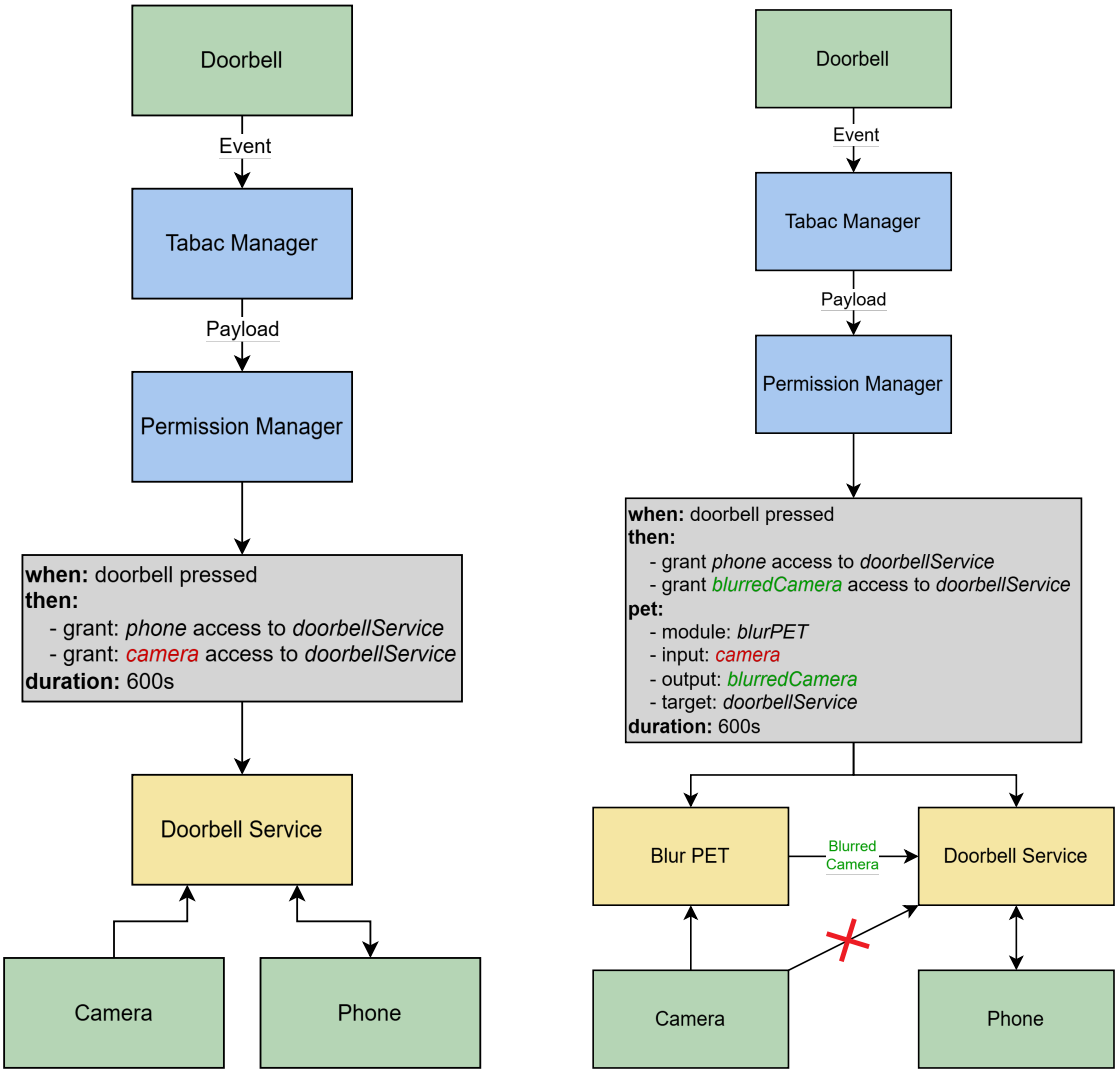
For a plain TABAC rule, `decodeActions()` produces one OpenHAB MQTT-publish action per concerned module: each module's permission grants are serialised and published to that module's individual authorisation topic, `admin/auth-<moduleId>` (dashes in the module ID are replaced by underscores). For a rule that contains a `pet` block, the approach changes. Instead of publishing individual authorisation messages, `decodeActions()` assembles a single *activation envelope* that bundles together all permission grants, all flow values, and the full PET specifications produced by `TabacPet.getActivationSpec()`. This envelope is published as one message to the central `admin/rule-auth` topic, where the PermissionManager picks it up.

This design keeps the OpenHAB rule engine entirely unaware of PETs. From OpenHAB's perspective a rule simply fires a trigger and publishes an MQTT message; all privacy-specific logic lives in the PermissionManager. It is also the second place backward compatibility is preserved: the per-module versus bundled decision is made purely on whether the rule's PET list is non-empty, so plain rules still emit exactly the per-module `admin/auth-<moduleId>` messages they always did.

We also implemented the system this way to avoid having to encode complex rule-condition evaluation logic directly inside OpenHAB rules. Such logic can quickly become difficult to express, maintain, and scale as rules grow longer and more complex. By centralising PET-specific evaluation inside the PermissionManager,

the OpenHAB layer remains simple while the privacy logic stays modular and easier to evolve.

The difference between the two flows is illustrated in Figure 5.3.



(a) System flow without PET rules

(b) System flow with PET rules

Figure 5.3: System flow before and after PET integration

5.5 Permission Rewriting in the Permission Manager

The PermissionManager is the component that turns a PET specification into an enforced privacy property. In HubOS it already existed as a singleton that maintains, for each module, the set of time-bound permissions it currently holds, and that answers the `isServerPermitted` query used by the egress proxy. We extended it rather than replaced it: the original path that installs grants from a plain `admin/auth-<moduleId>` message is untouched, and we added a parallel path, `addRuleActivation()`, that handles the bundled `admin/rule-auth` envelope. The inbound MQTT dispatcher routes the two topics to the two paths, which is the third and final point where backward compatibility is preserved.

The core of the new path is the method that rewrites permissions for each PET. It first builds a map from module ID to the grants destined for that module. Then, for every PET specification, it locates the grant that would have given the `target` module access to the raw `source`, removes it from the map, gives the `source` grant to the `pet_module` instead, and adds to the `target` a grant for the `output` address. An implementation detail makes the guarantee robust rather than merely cosmetic: the target's raw-source grant is filtered out of the map *before* any permission is installed, so it never becomes an enforceable permission in the first place. The privacy property is therefore structural, the raw grant is never created, rather than something granted and then revoked.

Two further behaviours were needed to make this safe in practice. First, when a PET serves its output on a loopback address inside its own container, the target is not pointed at that container address directly; instead the PermissionManager rewrites the target's stream URL to the HubOS stream API on the host, so the target can reach the transformed output without ever holding a route to the PET container or to the raw source. Second, every entry point that installs a grant is guarded by an "is this module actually loaded?" check, so an activation that names a module which is not currently running is ignored rather than silently creating a dangling permission. The same logic prunes the permissions and cancels the expiry timers of modules that are no longer active when the set of running modules changes.

5.6 The Northbound MQTT Interface

Applications and PETs need to know what they are currently allowed to do, and they need a way to give access back when they no longer need it. We added a northbound MQTT interface for this. For each module, HubOS publishes on a per-module topic prefix (`hubos/v1/perm/<moduleId>/`) the events that matter to

the module: the current authorisation state, a notification when a grant is added or is about to expire, and a notification when it is revoked. A complementary *release* topic lets a module relinquish a grant early, before its period elapses, so that access is not held longer than it is actually used. Control commands to a PET, such as the instruction to start transforming a given stream for a given duration, are delivered on the module's supervision topic, and the PET reports progress on the status topic declared in the rule.

This interface also resolves a subtle ordering problem. A PET container connects to the broker as soon as it starts, but the MQTT ACLs that let it exchange control messages with a particular application are only granted when the corresponding rule fires, which may be later. A PET therefore re-subscribes to its command topics whenever it observes a permission event on its northbound topic, and an application that depends on a PET re-sends its start command after it sees the grant arrive. Keeping the communication layer decoupled from the permission logic in this way means new application and PET types can be added without changing the core system; it is also what allows a PET to be authorised dynamically rather than being trusted from the start.

The northbound interface is optional, and applications or PETs do not need to be modified to use the system. Developers can integrate with these topics if they wish to take advantage of dynamic permission notifications and early grant release, but existing components remain fully compatible without these additions.

5.7 PET Applications as Sandboxed Modules

We built three PET modules, a video-stream blur, a visual keyword watcher, and a voice scrambler, together with the third party applications that use them. Their privacy behaviour and the scenarios that motivate them are presented in the Use Cases chapter 6. What matters here is that they are ordinary HubOS applications and share a single, reusable implementation pattern. Each is a directory containing a manifest, a configuration file, and one or more modules, where every module ships its own `Dockerfile` and entry point. When HubOS launches a module it injects, as environment variables, the module's MQTT credentials and the names of its supervision, northbound and command/status topics, so the module needs no hard-coded knowledge of the broker layout.

Every PET, regardless of what it computes internally, speaks the same command/status contract over MQTT: it waits for a `start` command carrying an input URL, a duration, an output path and any transform-specific parameters, performs its transformation, exposes the result (a stream on an output path, or a series of detection events), and stops when told to or when its grant expires. The three PETs differ only in the transformation behind that contract, a Gaussian

or face-localised blur over an MJPEG stream, an FFmpeg audio-filter chain that anonymises a voice, or an object detector that emits events instead of video. Which is precisely what let us add new PET types without touching the rule engine. One small but necessary addition at the platform level was support for an application to be a pure PET provider: a module may now be flagged as a PET and ship without any TABAC rule of its own, since it is driven entirely by the rules of the applications that consume it.

5.8 Implementation Challenges

The extension did not come together in a straight line. Several parts were rebuilt more than once, and a number of problems only surfaced once the pieces ran together. We document the most significant difficulties here, because they shaped the final design.

A race in permission assignment. Once multiple applications were loaded together, modules could occasionally receive grants intended for another application. The active-module set was being rebuilt inside the per-application loop using a cumulative list, so a later application's activation could overwrite or leak into the permission map of an earlier one, and modules that had been removed left expiry timers running. The fix was to compute the active-module set once, to guard every grant path with an "is this module loaded?" check so activations for unknown modules are ignored, and to clear the timers of modules that are no longer active. This bug is the reason the structural guards described in Section 5.5 exist.

Designing the PET rule engine. The first attempt at a PET engine was a thin stub that did little more than log the PET's conditions and actions; it did not survive contact with the real requirements, which involve linking symbolic names, rewriting permissions before they are installed, and managing the lifecycle of the resulting grants. The engine was therefore redesigned and rewritten from scratch. The key design decision made during that rewrite was *where* the PET logic should live. We chose to keep OpenHAB completely unaware of PETs and to concentrate all PET handling in the PermissionManager, behind the single `admin/rule-auth` activation envelope. This kept the OpenHAB integration unchanged, which we inherited and did not want to destabilise, and confined the new complexity to one component. This is where we focused most of our designing efforts to build a system that is robust and compliant.

The streaming media pipeline. Getting a live video stream to flow into a sandboxed container, be transformed, and be re-served to a consumer was the single

most unstable part of the work, and the blur PET's media handling was rewritten several times. Approaches based on re-streaming with FFmpeg, on running a dedicated media server inside the container, and on an internal HTTP proxy were each tried and discarded before we settled on a Flask server that re-broadcasts a transformed MJPEG stream produced with OpenCV. A related problem was that the PET serves its output on a loopback address inside its container, which the consumer cannot and should not reach directly; this is what motivated proxying the output through the HubOS stream API, so that the consumer reaches the transformed stream through a host endpoint while never holding a route to the PET container or the raw source.

Enforcing end-to-end least privilege. Removing the target's raw-source grant is not by itself enough: the messages HubOS sends to the consumer must also never leak the raw source or the PET's internal details. Achieving this required reworking how flow messages are published so that a consumer receives only the routed output URL, the address it is allowed to use and none of the PET metadata or the raw source address. This was done incrementally, and an intermediate version that over-shared had to be walked back before the output-only behaviour was correct.

Chapter 6

Use Cases and Illustrative Examples

This chapter presents three concrete examples of the PET extension in HubOS. The goal is not only to show possible applications, but also to make clear how the extension behaves in a running system. Each case uses the same basic mechanism: a standard TABAC rule is extended with a PET rule that instructs HubOS how to route the data flow before permissions are granted.

A PET rule identifies the PET module to use, the raw data source, the protected output, and the target application. It may also define activation conditions, MQTT topics for control communication, and a configuration object. This keeps the extension close to the existing TABAC model: the usual trigger-condition-action structure remains unchanged, the PET part only adds routing information on top of it.

We implemented the examples below as HubOS applications and PET modules, and used them to validate the key parts of the extension: linking of module and server names, permission rewriting, MQTT access control and supervision, and revocation after the rule period expires.

6.1 Conditional Camera Access with Face Blurring - Transformation Approach

The first example is based on a smart doorbell installed at the entrance of a home. The home is equipped with HubOS, a doorbell trigger, and a network camera that monitors the entrance area. When somebody rings the doorbell, the user wants to receive a short camera clip through an external notification service, such as Telegram. This makes it possible to check the entrance remotely when the user is away from home.

This is a common smart home scenario, but it is also a good example of why binary access control is limited. The useful function is not simply "allow the application to access the camera". The useful function is more precise: allow the application to send a notification, and if video is needed, provide a version of the video that does not reveal more information than necessary.

6.1.1 Problem Statement

In a traditional smart home integration, the third-party application would either have access to the camera stream or it would not. Even if access is temporary, the permission usually still points to the raw stream. During the validity period of the rule, the application can observe everything visible in the camera feed. This is more data than the application needs to send a doorbell notification.

The problem becomes more visible when the camera may capture visitors, neighbors, delivery workers, or children passing in front of the house. Once the raw stream is sent to an external service, the local platform no longer controls how the image is stored, processed, or reused. The fact that access was granted only after a doorbell event does not solve this. The event limits when the stream is shared, but not what is shared.

6.1.2 Proposed Solution

The PET extension allows HubOS to separate these two questions: *when* the application may access data, and *what* it actually receives. The normal TABAC part of the rule still decides when the doorbell application may react. The PET part decides how the video stream is routed before it reaches the application. In this example, the raw camera feed is made available to a PET module called `stream_blur`. The doorbell handler, `ring_handler`, receives the protected output stream instead of the raw camera stream.

The implemented rule is triggered when `device.ringSwitchItem` is updated to `ON`. The rule grants the handler access to the Telegram server and to the protected camera output alias, `CameraUrlBlurred`. The PET rule links this output alias to the raw source `CameraUrl`. As a result, the target module is not given direct access to `CameraUrl`. The Permission Manager grants the PET module access to the raw camera stream and grants the handler access only to the transformed stream.

The PET condition is time-based. In the implemented rule, the blur policy is active outside the interval from 08:00 to 16:00. When this condition is active, HubOS starts the PET with a blur configuration. When the condition is not active, the route through the PET remains the same, but the PET is started with `blur_type` set to `none`. This detail is important because the target application does not need to switch between two different sources. It always receives the

protected output route, while HubOS decides whether the transformation itself should be active.

Figure 6.1 shows a doorbell notification received through Telegram when the face-blurring PET is active (left) and when it is disabled (right). When the PET is enabled the cloud service only receives the blurred video feed. In contrast, when the PET is disabled, the original unmodified video is sent. This example illustrates how the PET reduces the exposure of identifiable visual information to third-party services while preserving the functionality of the third party app.

Figure 6.2 illustrates the complete flow, from the doorbell trigger through permission rewriting to the final Telegram notification.

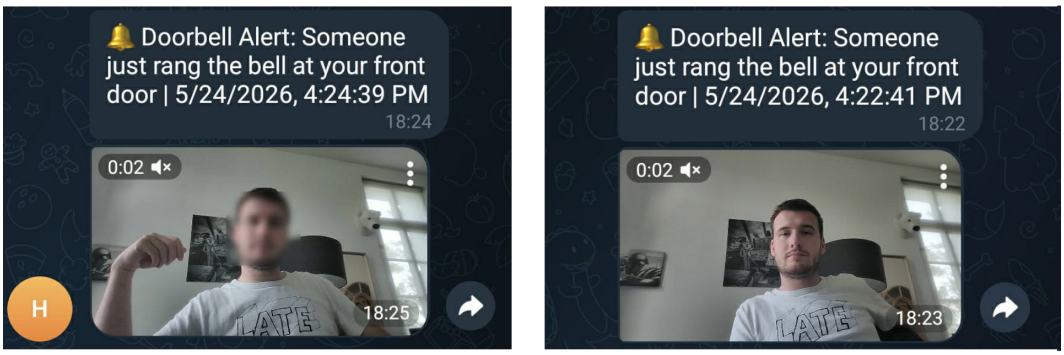


Figure 6.1: Comparison of a doorbell notification before (right) and after (left) the application of the face-blurring PET

6.1.3 Implementation

The implementation uses the existing TABAC pipeline as much as possible. The TABAC Manager parses the rule, links the names used in the rule to the configured modules and servers, and sends a rule activation message to the Permission Manager through the `admin/rule-auth` MQTT topic. This activation message contains both the ordinary permissions and the attached PET rule.

When the Permission Manager receives the activation, it builds the normal permission grants first. It then checks the PET rule. For a transparent PET route, it removes any direct target permission to the raw source, grants the PET module access to that raw source, and grants the target module access to the output alias. If the output is served locally by the PET container, for example on `127.0.0.1:8081`, HubOS does not expose the container address directly to the target. Instead, it exposes the stream through the HubOS API stream endpoint and sends this routed URL in the supervision message to the handler.

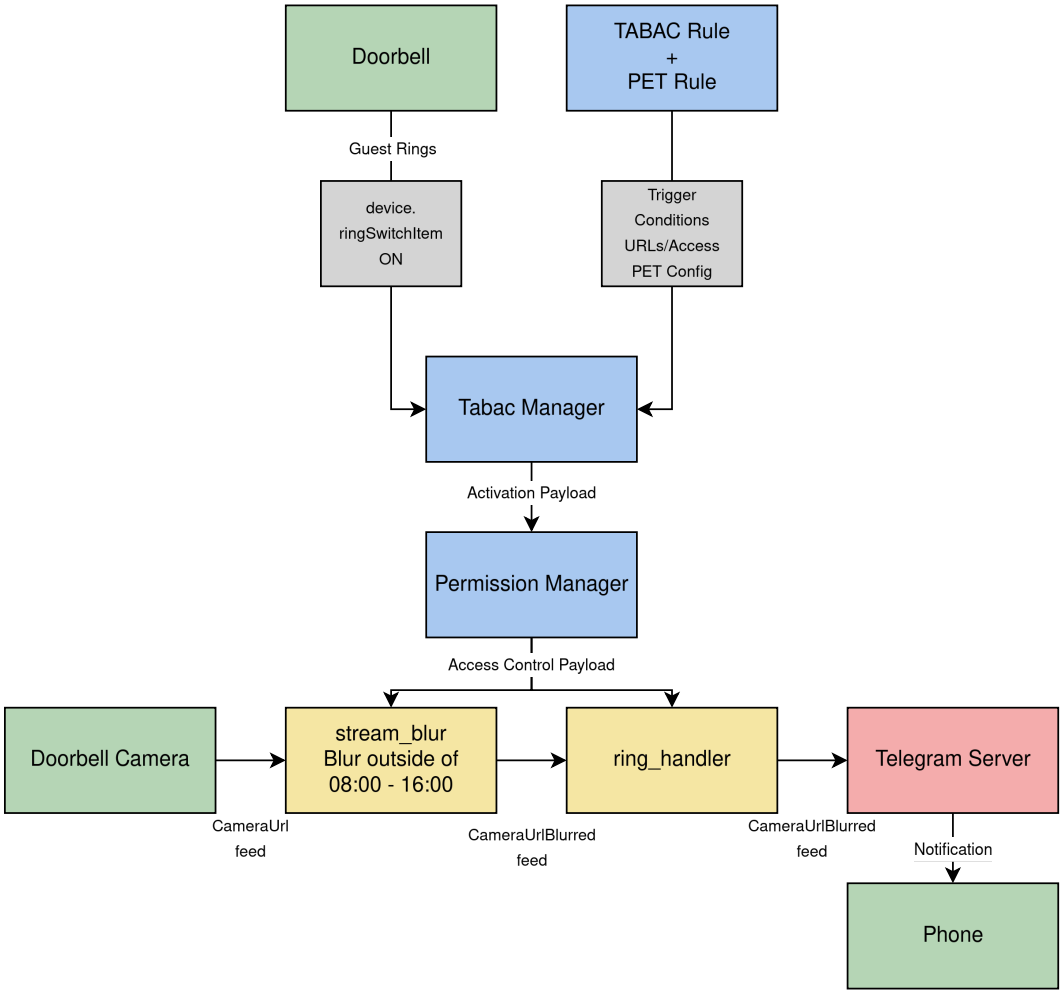


Figure 6.2: Doorbell use case

The `stream_blur` PET is implemented in Python with Flask and OpenCV. It listens for a start command, opens the configured camera stream, and serves an MJPEG output stream on the requested output path. The command contains fields such as `input_url`, `duration_seconds`, `output_path`, and `blur_type`. In the current prototype, the PET supports `full`, `faces`, and `none`.

The doorbell handler does not receive the PET rule itself. It receives the normal flow message, extended by HubOS with the routed `stream_url`. The handler then sends the notification through Telegram with that URL. From the handler's point of view, it is still reacting to a doorbell event and sending a camera link. The privacy-specific routing is enforced by HubOS and the Permission Manager, not by the application logic.

The MQTT permissions are also scoped to the PET rule. The rule declares

a command topic and a status topic for the blur PET. HubOS grants access to these exact topics only. Wildcard MQTT topics are rejected by validation, and the MQTT ACLs are removed when the rule period expires.

We encountered several challenges while implementing this PET, mainly due to hardware limitations. When the face blur is enabled, the system must first detect faces in the video stream and then blur them. However, scanning every frame for faces was not feasible on the target hardware because of performance constraints.

To improve performance, we reduced the detection frequency and analyzed only one frame out of every five. When a face is detected, the system applies the blur to the detected region for the following four frames. While this approach improves performance, it is not ideal when faces move quickly, as the blurred area may no longer accurately match the face position.

We also reduced the resolution used for face detection, which further improved performance at the cost of detection reliability. As a result, the feature works correctly but with lower precision than originally intended.

We discuss the benchmarks and performance implications of these design choices in Chapter 8.

6.2 Access to Local Inference Instead of Raw Video - Abstraction Approach

The second example considers a security application that wants to notify the user when a relevant object appears in a local camera feed. The application might be interested in a person, a pet, or another visual keyword. In the implemented prototype, this is represented by a PET called `keyword_watch` and a dashboard application called `keyword_pet_dashboard`.

The useful output of this scenario is not the video itself. The application only needs to know that a configured object was detected. For example, it may need to publish an event saying that a person was detected in the hallway, or that a cat was seen near the entrance.

6.2.1 Problem Statement

Granting a security application access to the raw camera feed would expose much more information than required. A raw indoor video stream may reveal the layout of the home, the habits of the occupants, private objects in the room, and moments that have no relation to the security function. Even if the application behaves correctly, the permission itself is unnecessarily broad.

Classical TABAC rules can restrict the duration of the permission, but they still grant access to a resource. In this use case, the better privacy property is

data minimization: the third-party application should receive only detection events, while the video processing remains local.

6.2.2 Proposed Solution

The PET extension can also support this style of privacy-preserving abstraction. Instead of forwarding a transformed video stream to the target application, HubOS authorizes a local PET to inspect the stream and authorizes the target application to communicate with the PET over declared MQTT topics. The target application does not need the camera URL. It only needs permission to send a command and to receive detection messages.

In the implemented rule, the dashboard sends a `START` event. The normal TABAC action grants the dashboard access to the protected output alias `KeywordWatchEvents` for a limited period and sends the flow value `PET_AUTHORIZED`. The PET rule connects the raw `ConfiguredVideoStream` source to the `keyword_watch` PET. It also declares the MQTT topics `pet/keyword_watch/command` and `pet/keyword_watch/status`.

This use case is less transparent than the camera blurring example. The dashboard must know how to speak to the PET. It sends a command containing the keywords to watch, the duration, and optional parameters such as confidence threshold or sampling interval. The PET then publishes status and detection messages. This makes the third-party application PET-aware, but it also gives it a much smaller and more meaningful interface than raw video.

While the implemented prototype focuses on the local dashboard interface, the same design naturally extends to external notification services. The `keyword_pet_dashboard` could forward detection events to a service such as Telegram, so that the user receives a phone notification when a keyword is detected. In this scenario, the cloud service would receive only a short text event, never the raw video feed, and never a transformed version of it. The privacy guarantee is stronger than in the camera blurring case: data minimization is enforced at the source, not merely at the transformation step.

Figure 6.3 illustrates this flow, from the local camera through the PET inference barrier to the detection events that reach the dashboard and, optionally, an external service.

6.2.3 Implementation

The `keyword_watch` PET is implemented as a local computer vision module. It opens the authorized video stream, samples frames, and checks them against the requested keywords. The detector can use an OpenCV DNN YOLO model when a local ONNX model is mounted. If no YOLO model is available, the prototype falls

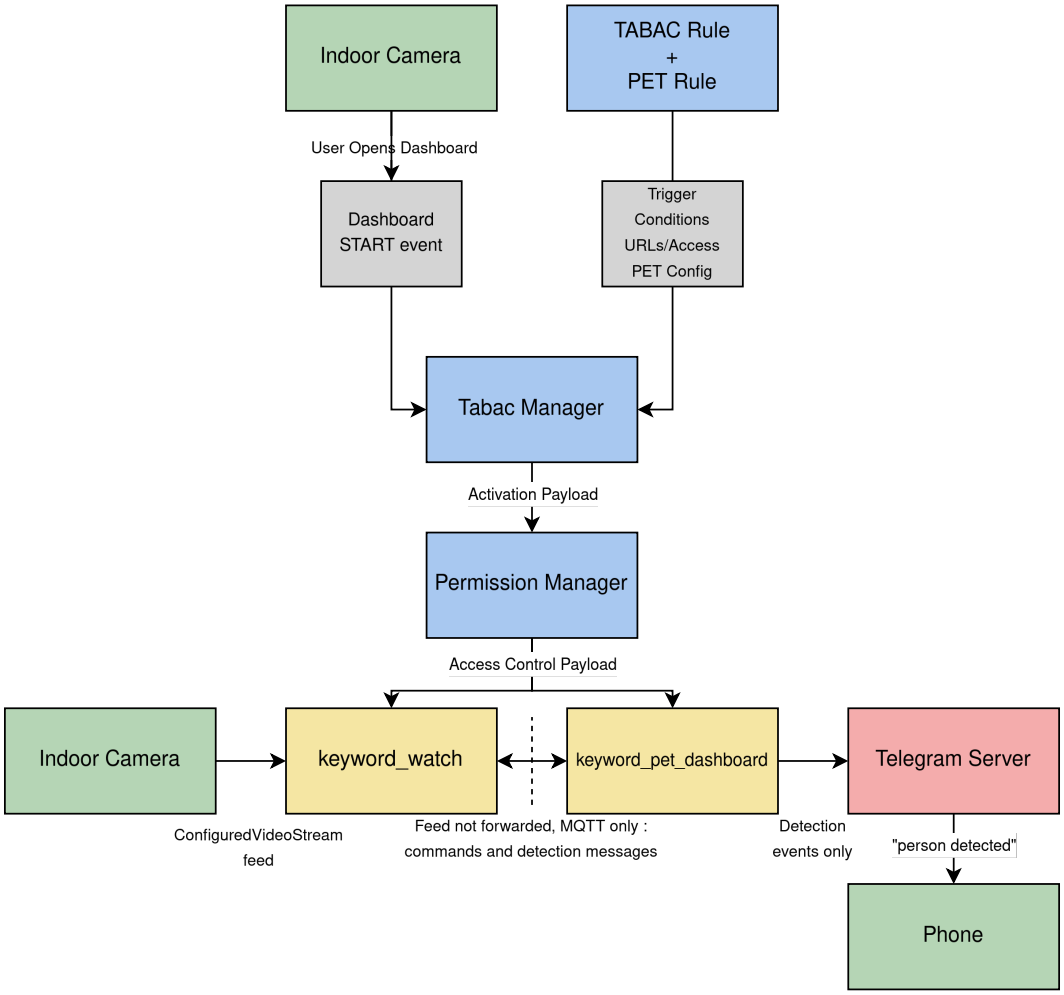


Figure 6.3: Keyword detection use case

back to OpenCV Haar cascades for a smaller set of keywords such as `cat`, `face`, and `person`. A mock detector is also available for MQTT API tests.

When a keyword is detected, the PET publishes a JSON message on its status topic. The message contains the event type, session identifier, keyword, detector label, confidence, bounding box when available, and timestamp. This means that the dashboard can react to the event without seeing the underlying frame. More complex aggregation, such as reporting a count of detections over a longer period, can be built on the same MQTT interface, but the implemented prototype focuses on event messages.

This example also shows why the PET rule contains an `autostart` configuration option. For the blur and audio examples, HubOS can start the PET directly because the transformation is known from the rule. For keyword watching, the dashboard

supplies the actual keywords at runtime. The rule therefore sets `autostart` to `false`. HubOS still grants the PET access to the raw stream and grants the dashboard the exact MQTT permissions, but the security application sends the start command itself after it receives the authorization flow.

The security property is different from the first example. The raw stream is still processed locally by the PET, because object detection requires access to the frames. However, the third-party dashboard never receives the raw stream. It receives a narrow event channel that contains only the result of the local inference.

6.3 Voice Anonymization Using a Voice Scrambler - Transformation Approach

The third example applies the same idea to audio. A smart home application may need to send a short voice or audio clip to an external service. The prototype uses an application called `AudioScrambleTelegram`, where an audio trigger causes a Telegram notification to be sent. The same pattern could also be used for voice assistants or other speech-driven services.

In this case, the data is sensitive for a different reason. A voice recording may reveal the speaker's identity, accent, emotional state, background sounds, and other private information. It may also be useful for later speaker recognition or voice cloning. The user may accept that a service receives speech content, but not that it receives a clean copy of the original voice signal.

6.3.1 Problem Statement

If the target application receives the raw microphone feed, the platform has little control over what happens after transmission. A time-limited permission is useful, but it does not change the nature of the data being shared. The raw audio still contains speaker identity features.

Another difficulty is that users may not want to configure a different privacy policy for every voice-related service. For many cases, the desired policy is stable: whenever this application receives microphone audio, it should receive an anonymized stream rather than the raw input.

6.3.2 Proposed Solution

The proposed solution is to route the microphone stream through a voice anonymization PET before it reaches the target module. The implemented rule attaches a PET called `audio_scrambler` to the audio notification application. The PET rule

has an empty condition array, which means that it applies whenever the main TABAC rule is triggered.

The normal rule grants the target handler access to Telegram and to the protected output alias `AudioUrlScrambled`. The PET rule links this protected output to the raw source `MicrophoneAudioUrl`. As in the video case, the Permission Manager grants the PET module access to the raw source and grants the target module access only to the transformed output. The target module is not granted the raw microphone stream. The resulting data flow mirrors Figure 6.2, with the microphone taking the place of the camera and `audio_scrambler` replacing the video blur PET.

This is a static PET policy. Unlike the camera example, it does not depend on time of day. Unlike the keyword detection example, the target application does not need to choose detection parameters at runtime. The rule itself contains the transformation settings, such as the scramble profile, output codec, sample rate, channel count, and duration.

6.3.3 Implementation

The `audio_scrambler` PET receives a start command from HubOS supervision. The command contains the input URL, the output path, the duration, and audio transformation settings. The implementation uses an FFmpeg pipeline to transform the audio stream. The available profiles are `light`, `standard`, and `strong`. Depending on the profile, the PET applies filters such as high-pass and low-pass filtering, rate shifting, resampling, tempo correction, tremolo, optional frequency-domain noise reduction, and gain normalization.

The goal is to reduce speaker identity cues while preserving the usefulness of the audio for downstream processing. This approach should not be interpreted as a cryptographic guarantee against all forms of forensic analysis. Rather, it is a practical privacy-enhancing measure that reduces the amount of directly identifiable voice information exposed to the target service.

We acknowledge that this may not be the most effective method for audio anonymization. In this work, it is used solely as a proof of concept. An ideal solution would rely on an audio transformation algorithm that provides strong speaker anonymization guarantees and makes voice cloning infeasible. However, the focus of this work is the implementation and integration of PETs, rather than the design of novel anonymization algorithms.

The target handler receives the routed output URL in its supervision message. It accepts fields such as `stream_url` or `audio_stream_url` and uses that URL when sending the Telegram alert. As with the camera example, the handler does not need to know the raw microphone URL. It only consumes the output that HubOS authorized.

The rule period also controls the lifetime of the PET-related permissions. If the normal action is valid for 600 seconds and the PET configuration asks for a shorter duration, the Permission Manager uses the shorter duration for the PET start command. When the permission expires, the normal permissions are revoked and the PET MQTT permissions are removed as well.

6.4 Comparison of the Two Approaches

The first and third examples illustrate a transformation approach. The original data flow is preserved at the level of data type: the application still receives video in the first case and audio in the third case. However, the data is modified before it reaches the application. The target module can continue to perform its task, but it receives a less sensitive version of the stream.

The second example illustrates an abstraction approach. The target application does not receive the original video stream, and it does not receive a visually transformed version of it either. Instead, it receives derived events produced by local inference. The PET consumes the raw stream locally and publishes only the information that is needed by the application.

These two approaches have different trade-offs. Transparent transformation is easier to integrate when an existing application already expects a stream. It can often be introduced by changing the TABAC rule and the routing configuration, while keeping the application interface mostly stable. Abstraction provides stronger data minimization, but it requires a more explicit interface between the application and the PET. The application must be designed to send commands and interpret PET results.

All three examples share the same enforcement principle. The target module is not trusted to ignore raw data. Instead, HubOS prevents the raw permission from being granted to the target in the first place. The PET module receives the raw source because it needs it to perform the privacy operation. The target module receives the protected output or the PET event channel. MQTT permissions are limited to the topics declared in the rule, and all grants remain temporary.

This is the main benefit of integrating PETs into the TABAC rule model. Privacy is no longer expressed only as an allow or deny decision. It can also describe the path that data must take, the module that must process it, the output that the target is allowed to consume, and the conditions under which the transformation should be active. The result is still compatible with existing TABAC rules, but it gives HubOS a way to enforce more precise and context-aware data sharing policies.

Chapter 7

Evaluation

This chapter evaluates the implemented system from three angles: whether it correctly enforces the behaviour it claims, whether the rule model is expressive enough to capture real privacy requirements, and whether the system is viable on the kind of hardware a typical home user would deploy.

7.1 Correctness of the PET Extension

The central claim of the PET extension is that when a rule with a `pet` block is triggered, the target module loses direct access to the raw data source and receives access only to the PET's output. This permission rewriting is the mechanism that makes the privacy guarantee concrete. If it fails, an application can simply bypass the PET entirely by connecting to the raw source it was originally authorised to access.

A set of unit tests covers the Permission Manager's behaviour for each relevant case. The tests verify that after a PET rule is activated, the target module holds a permission to the transformed output but not to the raw source, and that the PET module holds a permission to the raw source. We check this for both video stream routing and voice scrambling, which use slightly different connection topologies.

We tested two additional cases. When the PET output is served on a loopback address inside the PET container, the Permission Manager exposes it through the HubOS stream API rather than granting the target direct access to the container address. This prevents the target from reaching other services on the same loopback interface. When a rule references a module that is not loaded, the activation is ignored entirely rather than falling back to a permissive state.

We also verified the permission lifecycle: after the rule period expires, both the standard permissions and the MQTT permissions granted for PET control topics are revoked. A test with a one-second rule period confirms that the target loses

access to the output stream and that the PET's MQTT topics are removed at the same time.

Taken together, these tests give reasonable confidence that the core invariant holds: a target module cannot reach the raw source once a PET rule is active, and the access it does have expires on schedule.

7.2 Correctness of PET Conditions

PET conditions allow the system to apply different behaviour depending on runtime context. We currently only implemented time-based conditions: a PET can be activated only within or outside a specified time range.

We tested two cases. When the current time falls outside the allowed range, the PET is started with the configured transformation active and the target receives the transformed output. When the current time falls within the specified range, the routing still passes through the PET. However, in the case of the first example 6.1, the PET is configured with the parameter `blur_type` set to `none`. As a result, no anonymization is applied, while preserving the same processing pipeline and routing behavior. This means the target always connects to the same output address, regardless of whether the transformation is currently active. The application does not need to handle two different sources, and the system does not have to change the permission grants mid-rule. The PET itself becomes a pass-through when the condition is not met.

This behaviour was a deliberate design choice. Testing confirms that it is correctly implemented.

7.3 Isolation Properties

The functional evaluation confirms that the isolation holds for the three use cases. In the doorbell use case, the `ring_handler` module can only reach the Telegram server and the blurred camera output; it has no route to the raw camera stream. In the keyword detection use case, the dashboard module communicates with the `keyword_watch` PET only through the MQTT topics declared in the rule, not through any direct connection to the camera. MQTT access control is enforced separately: wildcard topic patterns are rejected at rule validation time, so a module cannot subscribe to more topics than the rule explicitly allows.

These properties were verified by running the use cases on the prototype and confirming that modules respect the boundaries set by the active rules, both by checking the granted permissions and by observing that attempts to access unauthorised resources fail at the network level.

7.4 Expressiveness

The rule model supports two distinct patterns for applying PET-based privacy control, discussed and compared in the previous chapter: transparent stream transformation and event abstraction. Both are expressed within the same rule format. The difference lies in whether the PET's output is a stream or a set of MQTT topics, and in whether the PET is started automatically by HubOS or manually by the target application. The `autostart` configuration option controls this: when set to `false`, HubOS grants the permissions but leaves the application responsible for sending the start command to the PET once it is ready.

What the model does not currently support is conditions based on sensor state or device context, as we only implemented time-based conditions. A rule cannot yet say, for example, "apply the blur only when no one is home" or "skip the voice scrambling when the user's phone is on the local network". These are meaningful use cases, and the structure of the condition format is already designed to accommodate them, but we did not implement the evaluation of non-time conditions.

7.5 Backward Compatibility

All existing TABAC rules without a `pet` field continue to work without modification. The `pet` field is optional, and its absence is handled the same way as before the extension was introduced. This was verified by running the existing test suite for the Permission Manager alongside the new PET tests, confirming that no regression was introduced.

7.6 Viability on Local Hardware

A practical question runs alongside correctness and expressiveness: is the system viable on the kind of hardware a typical home user would deploy? The benchmarks and deployment analysis in Chapter 8 provide the full measurements. The conclusion is that the system is viable for lightweight and audio workloads on standard home hardware, but video-intensive PETs, particularly those relying on YOLO-based inference such as the keyword detection use case, saturate a general-purpose CPU and require dedicated edge inference hardware to be practical. These constraints are discussed in detail in Section 8.8.

Chapter 8

Discussion

This chapter steps back from the implementation and looks at the broader picture: what the approach gets right, where it makes deliberate trade-offs, and what questions it does not yet answer.

8.1 Strengths

The core contribution of this work is the shift from binary access control to something more expressive. Instead of asking only whether a module may access a resource, our approach now also asks how the data should be handled when it does. This matters because in practice, most privacy problems in smart homes are not about access being granted to the wrong party altogether, but about too much data being shared with a party that legitimately needs some of it.

Integrating PETs directly into the TABAC rule model keeps this mechanism close to where access decisions are already made. A developer adding privacy control to their application does not have to build a separate pipeline outside the rule system. They add a `pet` block to their rule file, point it at a PET module, and the routing and permission rewriting happen automatically. The rest of the rule, including its trigger, conditions, and actions, stays unchanged.

The system is also backward compatible. Rules without a `pet` field continue to work exactly as before. This matters for adoption: existing applications do not break when the framework is updated, and developers can introduce PET-based protection incrementally.

8.2 What This Work Does Not Cover

It is important to be clear about the boundaries of what we built.

Our system operates inside a trusted hub. We assume that the hub itself is under the user's control and that the underlying operating system has not been compromised. If an attacker gains root access to the machine, none of the permission rewriting or MQTT access control we describe provides any real protection, because those mechanisms run on the same system. This is not a flaw specific to our design. It is a fundamental limitation of any software-only privacy system, and we do not claim otherwise.

What we do provide is isolation at the application level. Third-party modules run in containers and cannot directly reach raw data sources if a PET rule is active. If a module misbehaves or is malicious, it cannot bypass the network-level controls our system enforces to access the raw stream it was not granted. This is meaningful protection against a poorly written or mildly malicious application, even if it offers no protection against an attacker who controls the host.

Remote access is also outside the scope of this work. Users who want to access their smart home from outside their local network still need to set up a VPN or a similar mechanism. Integrating remote access securely into the local-first model is a real problem, but it is a separate one.

8.3 Limitations as Design Trade-offs

Every design choice in this work involves a trade-off. The limitations described below are not oversights; they are the known costs of the approach we chose.

8.3.1 Some data exposure remains

PETs reduce how much sensitive information leaves the home, but they do not eliminate data sharing entirely. Many useful applications genuinely require sending some data to an external service. A Telegram notification requires a connection to Telegram's servers. A voice assistant that cannot reach any external service is not very useful.

The goal was never to block all outbound data. It was to give the user control over what is sent and in what form. Some exposure is the price of functionality, and our system makes that trade-off explicit and configurable rather than invisible.

8.3.2 Configuration requires some technical understanding

Writing a TABAC rule with a PET block is not trivial. A user who wants to set up the doorbell blurring use case needs to know which module handles the camera, what the PET module is called, which server name maps to the raw stream, and

what condition they want to apply. This requires a level of familiarity with the system that most end users do not have.

The community store partially addresses this. When an application is distributed through the store, it comes with a ready-made rule file that already defines the PET configuration. A user installing the RingCamera application gets a working blur rule out of the box. They can use it as-is, or replace it with a community-provided alternative that matches their preferences (see Section 4.5). They never have to write a rule from scratch unless they want to.

Still, this is not a fully solved problem. A user who wants to understand what a rule actually does, or who wants to adapt it for their setup, needs to be able to read JSON and understand the basic concepts of triggers, conditions, and data routing. Better tooling and user interface support would help.

The prototype includes a visualization interface for TABAC. Figure 8.1 shows the page that lists the rules currently installed on the system, while Figure 8.2 shows the page that displays the active permissions granted by those rules. This kind of interface can make the configuration more approachable by surfacing rule intent and granted access in a form that users can inspect without reading raw JSON.

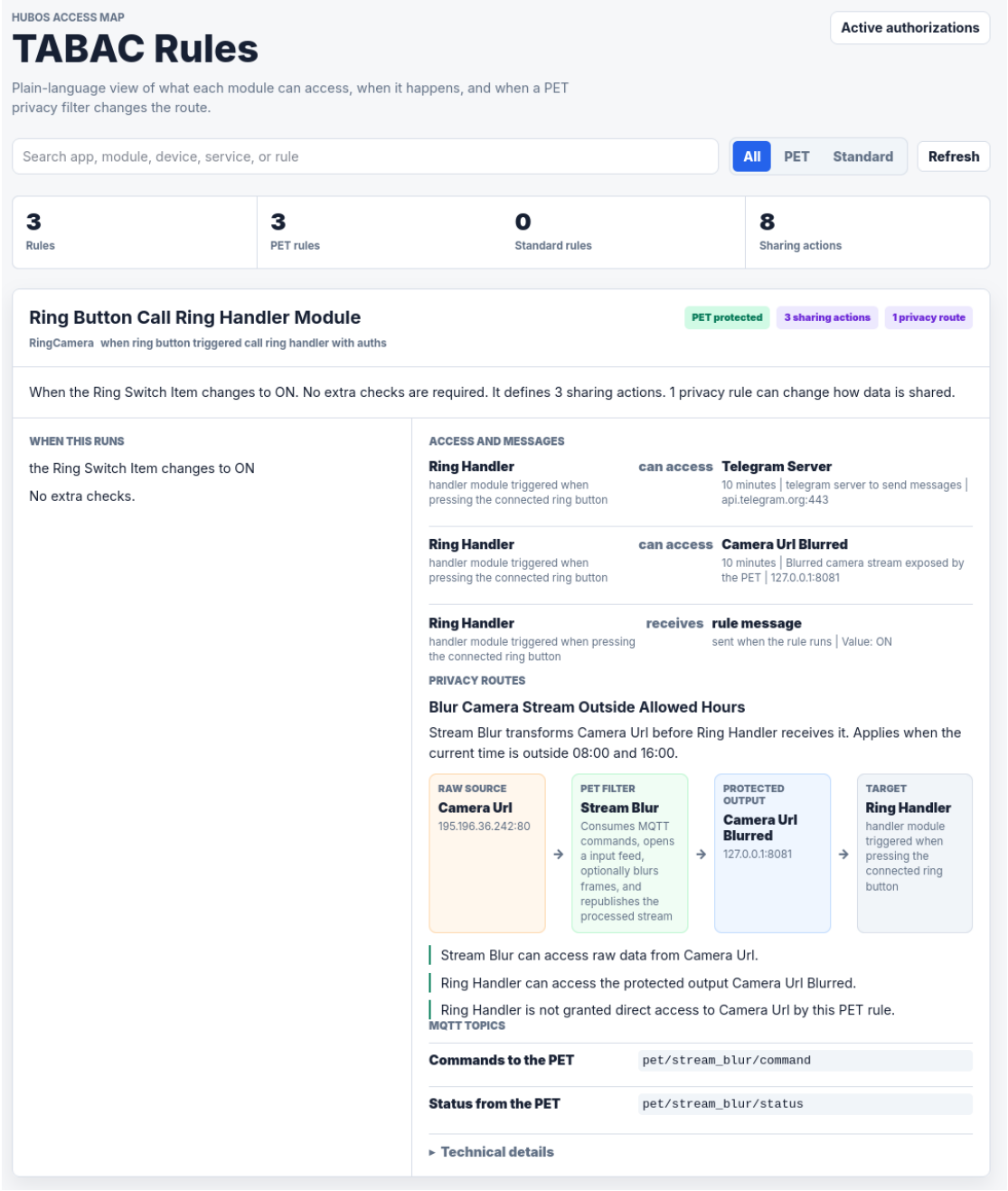


Figure 8.1: Prototype visualization of the TABAC rules installed on the system.

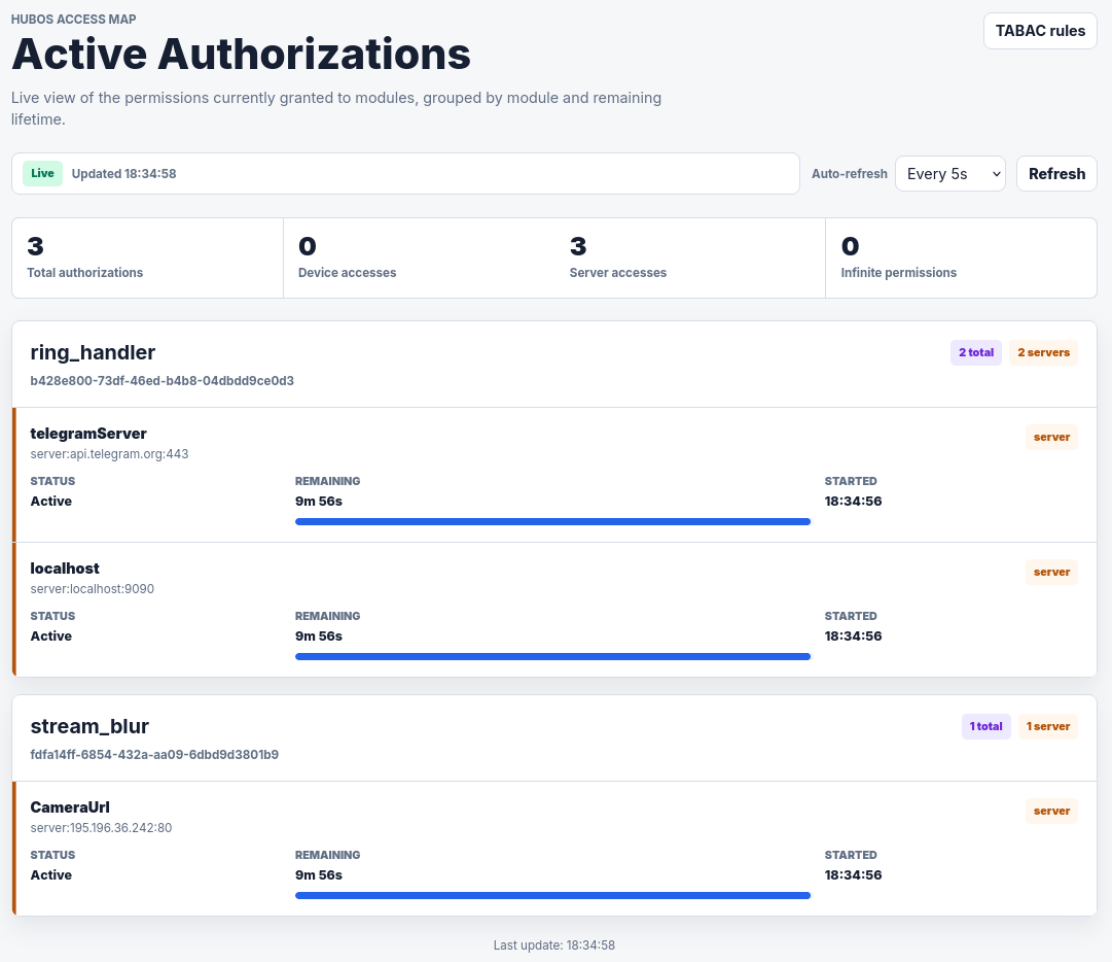


Figure 8.2: Prototype visualization of the active permissions granted by the current rules.

8.3.3 Trust in an open ecosystem

The community store model introduces a trust problem that is common to any open distribution system. Applications and rule files are contributed by third-party developers. A malicious or poorly written application might request more permissions than it needs, or a community rule file might silently weaken the privacy protections of a well-intentioned one.

There is no perfect solution to this. Code review, digital signatures, and reputation systems are the standard tools for managing trust in open ecosystems, and they would apply here as well. What the system already provides is that the rule file is separate from the application code and can be inspected, replaced, or overridden independently. A user does not have to trust the application developer's

rule file if they prefer a community-audited version. That is a better baseline than systems where permissions are hardcoded into the application and not accessible to the user at all.

8.4 The "Accept Everything" Problem

One question worth addressing is what happens if users simply accept every rule without reading it. This is not a hypothetical concern. A privacy framework that depends on informed user decisions is only as good as the decisions users actually make.

The prototype already highlights this limitation. While it can visualise installed rules and active permissions, it does not yet provide a clear warning when a rule requests broader access than usual, nor does it flag a PET configuration that applies weaker protections than the default. A user installing an application and accepting its rule file still has to trust the developer's description of what it does.

Addressing this would require work at two levels. At the interface level, richer rule summaries in natural language, visual indicators of privacy impact, and explicit alerts for potentially broad or weak protections would help users make more informed choices. At the ecosystem level, a baseline set of community-reviewed rules for common application types could give users a trustworthy starting point, similar to how curated app stores provide a degree of vetting before an application reaches users. Neither of these is technically hard. They are primarily design and community-building problems.

8.5 The Ecosystem and the Community Store

A fully open store where anyone can publish an application and a rule file is flexible and encourages contribution, but it makes trust harder to establish. A curated store with review requirements provides better quality signals, but it creates a bottleneck and may slow down the ecosystem. In practice, most successful open-source ecosystems have landed somewhere in between: open contribution with community review, signed packages, and reputation signals rather than centralised gatekeeping.

Another question is rule versioning and compatibility. As the TABAC rules model evolves, the rule format may change. Applications that were distributed with a rule file for an older version of the format need a migration path. This is a relatively standard software problem, but it is worth planning for early in the ecosystem's development, before a large number of deployed rule files have accumulated.

The marketplace model also raises questions about what counts as a PET. In the current implementation, PETs are Docker containers that receive a start command over MQTT and serve a transformed stream. This is a workable interface, but it is specific to streaming data. Other types of privacy-preserving transformations, such as aggregating telemetry over time or anonymising structured event data, would need a different interface. Defining a broader and more general PET contract is an interesting open problem.

8.6 Towards More Advanced Privacy Techniques

The architecture described in this work is compatible with more advanced privacy techniques that go beyond simple data transformation.

Differential privacy, for example, could be implemented as a PET that adds calibrated statistical noise to numeric sensor readings before forwarding them to an application. The application would receive data that is statistically useful but cannot be used to reconstruct individual readings with high precision. The rule model would not need to change: the user would simply attach a differential privacy PET to the relevant data flow. Implementing such a PET correctly requires care, in particular around the choice of the privacy budget and the sensitivity of the data, but the integration point is already there.

Similarly, a PET could apply techniques from federated learning by computing a local model update from sensor data and forwarding only the update to an external aggregation service, rather than the raw data. The hub would never send the underlying measurements outside the local network.

These directions are outside the scope of this thesis. They would each require their own study, both in terms of the correct use of the underlying techniques and the usability implications for the hub operator. But they illustrate that the architecture is not a dead end. The PET abstraction is general enough to accommodate them without changes to the core rule model.

8.7 Benchmarks

We measured CPU and memory usage for three scenarios: (1) the third-party app running without a PET, (2) the same app while a PET performed a transparent passthrough, and (3) the app while a PET applied a transformation. The Ring camera and voice recorder experiments were collected independently, and we also repeated the test with both apps active to confirm system-level concurrency effects. In the PET-enabled scenarios, the PET was instructed to process/forward the data stream for 30 seconds and was authorized to access the raw video stream for 40

seconds. Each plot tracks host CPU utilization (percent) and resident memory usage (MB) over elapsed time. The red dashed line marks the moment when the TABAC rule fired and the third-party app began its supervised operation.

Figure 8.3 shows two important effects for the Ring camera case: first, the initial capture-and-upload phase generates a sharp CPU spike because the app decodes, packages, and transmits video frames. Second, when a PET is present the host sustains a higher load for longer. A PET passthrough adds overhead due to the extra container and the additional stream forwarding path, while a PET transformation increases CPU usage further because the video frames are modified before being forwarded. Memory usage is higher in both PET cases because the PET runs in an isolated container and holds its own runtime state for the duration of the operation.

What is not visible in the graphs, however, is the degradation in the resulting video stream after PET processing. The processed video exhibited stuttering and missing frames. This behavior is caused by system overload, where the host is unable to simultaneously process the incoming video feed, perform face detection, apply the blur transformation, and output the transformed stream in real time. Despite our efforts to optimize the performance of the `videoBlurPet`, the computational cost of the transformation pipeline remained significant on the evaluated hardware.

In contrast, Figure 8.4 highlights the lower computational cost of audio handling in this prototype. The PET passthrough and PET transformation traces remain close together: the audio processing task is lightweight, so the primary overhead is again containerization and stream forwarding rather than expensive inference, and the PET container itself is the dominant cost. The host still allocates more memory once the PET container starts, but the CPU profile is more modest and the active period is shorter than for the video workload.

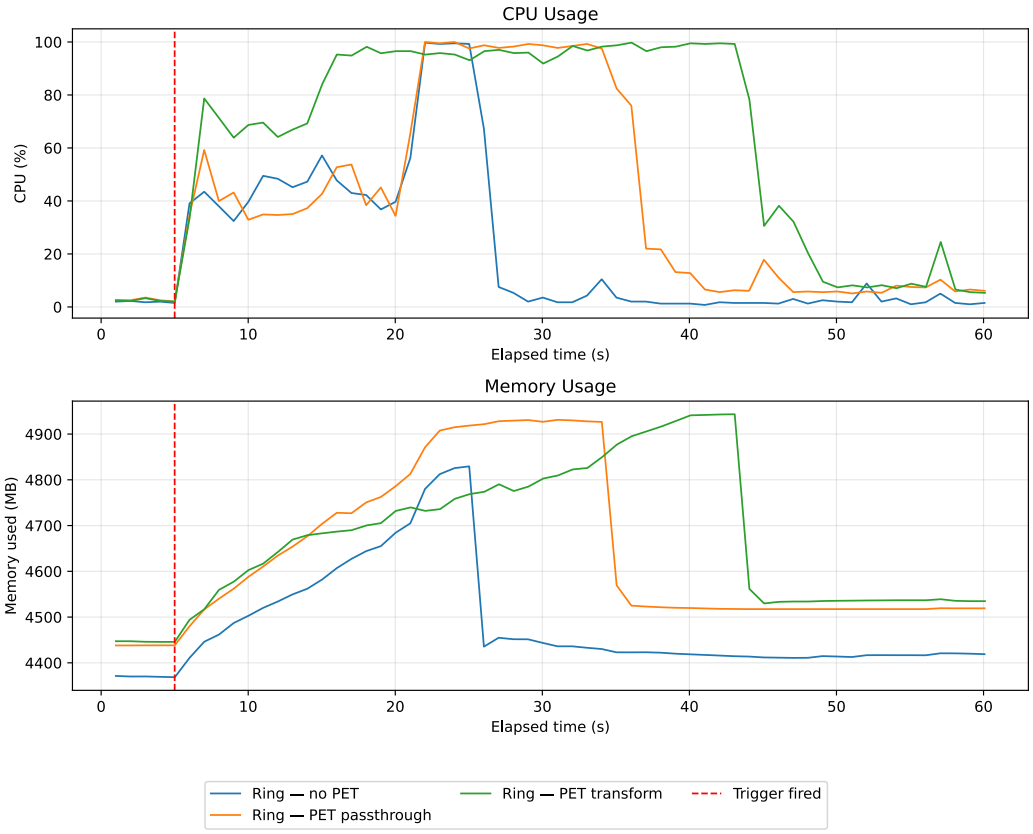


Figure 8.3: Ring camera: host CPU and memory usage.

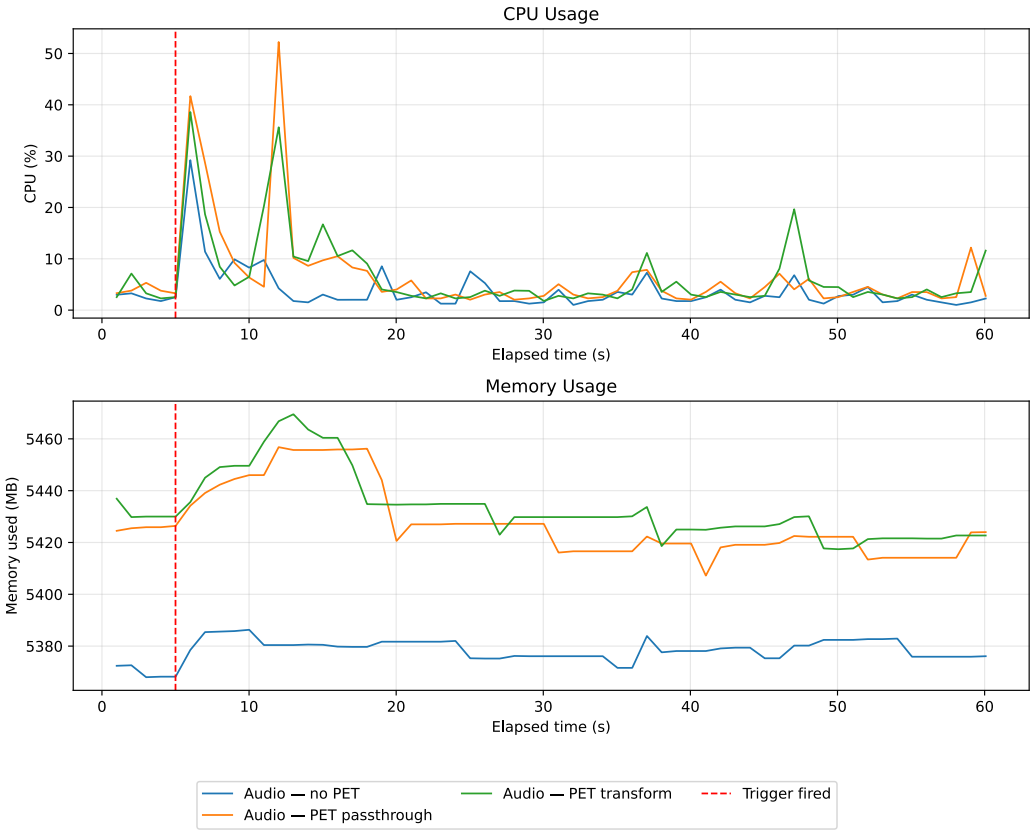


Figure 8.4: Voice recorder: host CPU and memory usage.

8.7.1 Observations

- CPU and memory overheads scale with PET complexity and concurrency.
- For video workloads, PETs add both higher peaks and a longer sustained CPU load because the stream is transformed or forwarded continuously.
- For the audio workload, CPU overhead remains relatively small, and the dominant cost is containerization rather than per-sample processing.
- Passthrough PETs add overhead mainly from containerization and additional network/I/O paths; transformation PETs add extra CPU cost when they manipulate the stream.
- Ensuring models are available locally and adding hardware acceleration (GPU/inference chip) greatly improves practicality on modest hubs.

8.8 Deployment and Viability

We deployed the prototype on an Intel NUC (x86, 4 cores, 8 GB RAM). This hardware is sufficient for lightweight workloads: the voice scrambling use case completes within expected resource bounds and the PET overhead does not affect responsiveness. However, it quickly becomes a bottleneck for more demanding workloads, such as CPU-intensive video processing or running multiple PETs concurrently.

The face-blurring use case illustrates this directly. Continuous frame decoding, face detection, and re-encoding saturate the CPU under sustained load, leading to dropped frames and stuttering. The bottleneck is the transformation pipeline, not the rule enforcement mechanism.

The keyword detection use case is the most demanding. As it performs machine-learning inference, performance depends heavily on whether an appropriate model is available locally. In our prototype, the `keyword_watch` PET targets YOLO inference, which is too slow on a general-purpose CPU without acceleration, but falls back to OpenCV Haar cascades when no YOLO model is mounted, which are less accurate. Running keyword detection and video blurring concurrently is not practical on the evaluated hardware.

These are deployment constraints, not fundamental limitations of the approach, so the privacy guarantees hold regardless of hardware. A natural path forward is dedicated edge inference hardware. NVIDIA Jetson devices[1] offer a full GPU environment in a low-power form factor, which would make sustained YOLO inference practical. Google Coral accelerators[2] are an alternative for lower power budgets, targeting TensorFlow Lite models through a dedicated TPU.

Power consumption is also a concern. A continuously running hub that processes video streams consumes significantly more energy than one handling only lightweight events. For energy-sensitive or low-power deployments, always-on PET processing may be impractical without hardware acceleration or careful duty cycling.

Verdict. The system is viable for lightweight and audio workloads on standard home hardware. Video-intensive PETs and ML-based inference require dedicated hardware acceleration to deliver acceptable performance.

[1] https://www.nvidia.com/fr-fr/autonomous-machines/embedded-systems/
[2] https://www.coral.ai/

Chapter 9

Conclusion

This chapter closes the thesis. We summarise the contributions of this work, identify directions for future research, and offer some final remarks on the broader context of the problem we addressed.

9.1 Summary of Contributions

Smart home devices are useful, but the way most of them work today asks users to accept a deal they rarely see clearly: functionality in exchange for continuous data collection by manufacturers and third-party services. Local-first platforms like HubOS push back against this by keeping data processing on the hub, but they still rely on binary access control. A module either gets access to a resource or it does not. That model is too coarse for real privacy requirements.

This work extends HubOS with a mechanism that changes the question from *whether* a module may access data to *how* the data reaches it. We introduce Privacy-Enhancing Technologies as first-class components inside the TABAC rule system. When a rule is triggered, the hub can now intercept the relevant data flow, pass it through a PET, and forward only the transformed result to the requesting module. The target application never receives the raw data. This is enforced at the permission level: the Permission Manager rewrites the access grants so that the raw source is only reachable by the PET, not by the application itself.

The extension fits into the existing rule structure as an optional `pet` block. Developers familiar with TABAC rules can add PET-based data routing without learning a new system, and existing rules without a `pet` field continue to work unchanged. PET activation can be conditional, for instance applying a blur only outside certain hours, without changing what the application sees on its end.

Three use cases validate the approach: camera feed blurring for a doorbell notification service, keyword detection as a substitute for raw video access, and

voice scrambling before sending audio to an external service. Together they cover the two main patterns the system supports: transforming a stream before it reaches an application, and replacing a stream with derived events produced by local inference.

9.2 Future Work

Several directions are worth pursuing to build on this work.

User interface and rule transparency. The prototype includes basic pages for visualising installed rules and active permissions, but there is room to go further. A well-designed interface could present the privacy implications of a rule in plain language, flag configurations that request broader access than comparable applications, and give users a clear picture of what data is leaving their home and through which path. Making this information accessible without requiring users to read JSON would lower the barrier to informed use of the system.

Richer PET conditions and types. The current implementation supports time-based conditions for activating a PET. Extending this to sensor values, occupancy states, or combinations of conditions would make the rules more expressive. Similarly, the current PET interface is designed around streaming data. A more general contract that also covers structured event data and aggregated telemetry would allow PETs to be applied to a wider range of data flows.

User acceptance and default configurations. Chapter 8 raises the concern that users may accept rule files without reading them. Research into how to present rule content in ways that support genuine understanding, and into what sensible defaults look like for common application types, would make the privacy guarantees of the system more reliable in practice. Community review processes for rule files are a natural complement to this.

Stronger privacy techniques. The PET abstraction is compatible with more advanced techniques such as differential privacy and local model inference. A PET that adds calibrated noise to numeric sensor readings, or that computes a model update locally and forwards only that, would provide stronger privacy properties than simple stream transformation. These are well-studied techniques, and integrating them as PET modules would be a natural extension of the framework.

Hardware and performance. Running video processing PETs on low-power hardware is a real constraint. Exploring what PET workloads are feasible on common hub hardware, and where dedicated accelerators would make a practical difference, would help ground the deployment model in realistic expectations.

9.3 Final Remarks

The gap this work addresses is specific: smart home platforms that keep data local but still share it in raw form when third-party applications need access to it. We did not set out to solve every privacy problem in connected environments, and we did not. What we did is show that the rule model in a local-first smart home OS is a natural place to introduce data transformation, and that doing so does not require redesigning the system from scratch.

The approach has real costs. It asks developers to declare their data flows upfront and accept that the hub may transform what their application receives. It asks users to understand, at some level, what a rule file means. These costs are worth accepting, but acknowledging them honestly is part of designing a system that can actually be deployed and trusted.

Privacy in smart homes is not a problem that gets solved once. New devices, new applications, and new threats will keep raising new questions. What a framework like this can offer is not a final answer, but a foundation that makes those questions easier to reason about and act on.

9.4 Usage of Artificial Intelligence

Artificial intelligence tools, including Claude and ChatGPT, were used to assist in rewriting and improving the clarity of some parts of this document. The ideas, design decisions, and implementation presented in this work are entirely our own. All technical choices and conclusions were made independently by the authors.

9.5 Acknowledgments

First and foremost, we would like to express our sincere gratitude to our supervisor, Professor Etienne Riviere, whose guidance, insight, and continuous support throughout this project were invaluable and greatly enriched our understanding of the subject. We also wish to thank our readers and jury members, Dr. Annanda Rath and Julien Gourgue, for taking the time to review and assess our work. Special thanks go to Valentin Kola for his time, contributions, and work on HubOS, as well as his help and support in navigating and understanding the system our

thesis builds upon. Finally, we would like to thank our families and friends for their support and encouragement thoughout this work. This project would not have been possible without the contributions and support of all these individuals, and we are deeply grateful for their involvement in our journey.

Bibliography

[1] I. Zavalvshyn, A. Legay, A. Rath, and E. Rivière, "Local-First Smart Home Applications with HubOS," in *2025 20th European Dependable Computing Conference (EDCC)*, Lisboa, Portugal: IEEE, Apr. 8, 2025, pp. 18–26, ISBN: 979-8-3315-1280-4. DOI: 10.1109/EDCC66201.2025.00015. Accessed: Sep. 24, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/11107379/.

[2] L. Rocher, J. Hendrickx, and Y.-A. de Montjoye, "Estimating the success of re-identifications in incomplete datasets using generative models," *Nature Communications*, vol. 10, 2019. DOI: 10.1038/s41467-019-10933-3.

[3] Y. Jia *et al.*, "Burglars' iot paradise: Understanding and mitigating security risks of general messaging protocols on iot clouds," *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 465–481, 2020. DOI: 10.1109/sp40000.2020.00051.

[4] "Home assistant," Accessed: Apr. 23, 2026. [Online]. Available: https://www.home-assistant.io/.

[5] "Openhab," Accessed: Apr. 23, 2026. [Online]. Available: https://www.openhab.org/.

[6] Z. N. Mohammad, F. Farha, A. O. Abuassba, S. Yang, and F. Zhou, "Access control and authorization in smart homes: A survey," *Tsinghua Science & Technology*, vol. 26, pp. 906–917, 2021. DOI: 10.26599/tst.2021.9010001.

[7] S.-C. Cha, T. Hsu, Y. Xiang, and K.-H. Yeh, "Privacy enhancing technologies in the internet of things: Perspectives and challenges," *IEEE Internet of Things Journal*, vol. 6, pp. 2159–2187, 2019. DOI: 10.1109/jiot.2018.2878658.

[8] C. Li and B. Palanisamy, "Privacy in internet of things: From principles to technologies," *IEEE Internet of Things Journal*, vol. 6, pp. 488–505, 2018. DOI: 10.1109/jiot.2018.2864168.

[9] N. Kaaniche, M. Laurent-Maknavicius, and S. Belguith, "Privacy enhancing technologies for solving the privacy-personalization paradox: Taxonomy and survey," *J. Netw. Comput. Appl.*, vol. 171, p. 102 807, 2020. DOI: 10.1016/j.jnca.2020.102807.

[10] A. Jacobsson, M. Boldt, and B. Carlsson, "A risk analysis of a smart home automation system," *Future Gener. Comput. Syst.*, vol. 56, pp. 719–733, 2016. DOI: 10.1016/j.future.2015.09.003.

[11] I. Zavalyshyn, A. Legay, A. Rath, and E. Rivière, "SoK: Privacy-enhancing Smart Home Hubs," *Proceedings on Privacy Enhancing Technologies*, vol. 2022, no. 4, pp. 24–43, Oct. 2022, ISSN: 2299-0984. DOI: 10.56553/popets-2022-0097. Accessed: Sep. 29, 2025. [Online]. Available: https://petsymposium.org/popets/2022/popets-2022-0097.php.

[12] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, "Local-first software: You own your data, in spite of the cloud," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2019, New York, NY, USA: Association for Computing Machinery, Oct. 23, 2019, pp. 154–178, ISBN: 978-1-4503-6995-4. DOI: 10.1145/3359591.3359737. Accessed: Sep. 29, 2025. [Online]. Available: https://doi.org/10.1145/3359591.3359737.

[13] S. Fu and S. Ratnasamy, "dSpace: Composable Abstractions for Smart Spaces," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, Virtual Event Germany: ACM, Oct. 26, 2021, pp. 295–310, ISBN: 978-1-4503-8709-5. DOI: 10.1145/3477132.3483559. Accessed: Sep. 29, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3477132.3483559.

[14] I. Zavalyshyn, N. O. Duarte, and N. Santos, "HomePad: A Privacy-Aware Smart Hub for Home Environments," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, Oct. 2018, pp. 58–73. DOI: 10.1109/SEC.2018.00012. Accessed: Sep. 29, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/8567657.

[15] H. Jin, G. Liu, D. Hwang, S. Kumar, Y. Agarwal, and J. I. Hong, "Peekaboo: A hub-based approach to enable transparency in data processing within smart homes," *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 303–320, 2022. DOI: 10.48550/arxiv.2204.04540.

[16] H. Chi, Q. Zeng, X. Du, and L. Luo, "Pfirewall: Semantics-aware customizable data flow control for smart home privacy protection," *ArXiv*, vol. abs/2101.10522, 2021. DOI: 10.14722/ndss.2021.24464.

[17] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, vol. 29, no. 2, pp. 38–47, 1996. DOI: 10.1109/2.485845.

[18] V. C. Hu *et al.*, "Guide to attribute based access control (abac) definition and considerations," National Institute of Standards and Technology, NIST Special Publication 800-162, 2015. DOI: 10.6028/NIST.SP.800-162.

[19] S. Ameer, J. O. Benson, and R. Sandhu, "Hybrid approaches (abac and rbac) toward secure access control in smart home iot," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, pp. 4032–4051, 2023. DOI: 10.1109/tdsc.2022.3216297.

[20] A. Ansari, M. Nazir, and K. Mustafa, "Ontology-based classification and detection of the smart home automation rules conflicts," *IEEE Access*, vol. 12, pp. 85 072–85 088, 2024. DOI: 10.1109/access.2024.3415632.

[21] C. Nandi and M. D. Ernst, *Automatic Trigger Generation for Rule-based Smart Homes*. Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, 2016. DOI: 10.1145/2993600.2993601.

[22] J. Quantrill, N. Khajehnouri, and M. H. Alalfi, "Ohit - a framework for openhab interaction threats identification in iot systems," in *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2025, pp. 1047–1056. DOI: 10.1109/compsac65507.2025.00136.

[23] A. Qashlan, P. Nanda, X. He, and M. Mohanty, "Privacy-preserving mechanism in smart home using blockchain," *IEEE Access*, vol. 9, pp. 103 651–103 669, 2021. DOI: 10.1109/access.2021.3098795.

[24] S. Dutta, S. S. L. Chukkapalli, M. Sulgekar, S. Krithivasan, P. K. Das, and A. Joshi, "Context sensitive access control in smart home environments," *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pp. 35–41, 2020. DOI: 10.1109/bigdatasecurity-hpsc-ids49724.2020.00018.

[25] W. He, N. Reitinger, A. Almogbil, Y.-S. Chiang, T. J. Pierson, and D. Kotz, "Contextualizing interpersonal data sharing in smart homes," *Proc. Priv. Enhancing Technol.*, vol. 2024, pp. 295–312, 2024. DOI: 10.56553/popets-2024-0051.

[26] S.-C. Cha, T.-Y. Hsu, Y. Xiang, and K.-H. Yeh, "Privacy Enhancing Technologies in the Internet of Things: Perspectives and Challenges," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2159–2187, Apr. 2019, ISSN: 2327-4662. DOI: 10.1109/JIOT.2018.2878658. Accessed: Sep. 30, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/8515008/references.

[27] Razi Qaiser and Piyush Raja, "Enhancing Data Privacy: A Comprehensive Survey of Privacy-Enabling Technologies," *ResearchGate*, Aug. 6, 2025. DOI: 10.1109/ACCESS.2025.3546618. Accessed: Sep. 29, 2025. [Online]. Available: https://www.researchgate.net/publication/389443355_Enhancing_Data_Privacy_A_Comprehensive_Survey_of_Privacy-Enabling_Technologies.

[28] G. Poh, P. Gope, and J. Ning, "Privhome: Privacy-preserving authenticated communication in smart home environment," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, pp. 1095–1107, 2019. DOI: 10.1109/tdsc.2019.2914911.

[29] N. Waheed, F. Khan, M. Jan, A. Z. Alalmaie, and P. Nanda, "Privacy-enhanced living: A local differential privacy approach to secure smart home data," *2023 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, pp. 1–6, 2023. DOI: 10.1109/coins57856.2023.10189261.

[30] M. Boteju, T. Ranbaduge, D. Vatsalan, and N. Arachchilage, "Sok: Demystifying privacy enhancing technologies through the lens of software developers," *ArXiv*, vol. abs/2401.00879, 2023. DOI: 10.48550/arxiv.2401.00879.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**